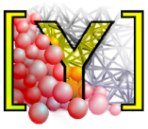


Yade Documentation (3rd ed.)

3rd Edition *based on Yade version 20261020-9630 a198ad4 forky1, October 20, 2026*

Authors

- **Václav Šmilauer** Freelance consultant (http://woodem.eu)
- **Vasileios Angelidakis** Newcastle University, UK
- **Emanuele Catalano** Univ. Grenoble Alpes, 3SR lab.
- **Robert Caulk** Univ. Grenoble Alpes, 3SR lab.
- **Bruno Chareyre** Univ. Grenoble Alpes, 3SR lab.
- **William Chèvremont** Univ. Grenoble Alpes, LRP
- **Sergei Dorofeenko** IPCP RAS, Chernogolovka
- **Jérôme Duriez** INRAE, Aix Marseille Univ, RECOVER, Aix-en-Provence, France
- **Nolan Dyck** Univ. of Western Ontario
- **Jan Eliáš** Brno University of Technology
- **Burak Er** Bursa Technical University
- **Alexander Eulitz** TU Berlin / Institute for Machine Tools and Factory Management
- **Anton Gladky** TU Bergakademie Freiberg
- **Ning Guo** Hong Kong Univ. of Science and Tech.
- **Christian Jakob** TU Bergakademie Freiberg
- **François Kneib** Univ. Grenoble Alpes, 3SR lab. / Irstea Grenoble
- **Janek Kozicki** Gdansk University of Technology
- **Donia Marzougui** Univ. Grenoble Alpes, 3SR lab.
- **Raphaël Maurin** Irstea Grenoble
- **Chiara Modenese** University of Oxford
- **Gerald Pekmezi** University of Alabama at Birmingham
- **Luc Scholtès** Univ. Grenoble Alpes, 3SR lab.
- **Luc Sibille** University of Nantes, lab. GeM
- **Jan Stránský** CVUT Prague
- **Thomas Sweijen** Utrecht University
- **Klaus Thoeni** The University of Newcastle (Australia)
- **Chao Yuan** Univ. Grenoble Alpes, 3SR lab.
- **Karol Brzeziński** Warsaw University of Technology

Citing this document

When referring to Yade-DEM software in scientific publication please cite it "by DOI" as follows: Šmilauer V. et al. (2021) Yade Documentation 3rd ed. The Yade Project. DOI:10.5281/zenodo.5705394. http://yade-dem.org

See also http://yade-dem.org/doc/citing.html.

Contents

1 Guided tour **1**

1.1 Introduction . . . 1
1.1.1 Getting started . . . 1
1.1.2 Architecture overview . . . 6
1.2 Tutorial . . . 16
1.2.1 Introduction . . . 16
1.2.2 Hands-on . . . 16
1.2.3 Data mining . . . 26
1.2.4 Setting up a simulation . . . 31
1.2.5 Advanced & more . . . 35
1.2.6 Examples with tutorial . . . 36
1.2.7 More examples . . . 51

2 Yade for users **57**

2.1 DEM formulation . . . 57
2.1.1 Collision detection . . . 57
2.1.2 Creating interaction between particles . . . 61
2.1.3 Kinematic variables . . . 63
2.1.4 Contact model (example) . . . 66
2.1.5 Motion integration . . . 66
2.1.6 Periodic boundary conditions . . . 74
2.1.7 Computational aspects . . . 79
2.2 User's manual . . . 80
2.2.1 Scene construction . . . 80
2.2.2 Controlling simulation . . . 100
2.2.3 Postprocessing . . . 117
2.2.4 Python specialties and tricks . . . 122
2.2.5 Extending Yade . . . 123
2.2.6 Troubleshooting . . . 123
2.2.7 Getting in touch with Yade community . . . 124
2.3 Yade wrapper class reference . . . 125
2.3.1 Bodies . . . 125
2.3.2 Interactions . . . 126
2.3.3 Global engines . . . 127
2.3.4 Partial engines . . . 127
2.3.5 Dispatchers . . . 127
2.3.6 Functors . . . 127
2.3.7 Bounding volume creation . . . 127
2.3.8 Interaction Geometry creation . . . 127
2.3.9 Interaction Physics creation . . . 127
2.3.10 Constitutive laws . . . 134
2.3.11 Internal forces . . . 134
2.3.12 Callbacks . . . 136
2.3.13 Preprocessors . . . 136
2.3.14 Rendering . . . 136
2.3.15 Simulation data . . . 137

2.3.16 Other classes . . . 145
2.4 Yade modules reference . . . 145
2.4.1 yade.bf module . . . 145
2.4.2 yade.bodiesHandling module . . . 147
2.4.3 yade.export module . . . 148
2.4.4 yade.geom module . . . 153
2.4.5 yade.gridpfacet module . . . 157
2.4.6 yade.libVersions module . . . 161
2.4.7 yade.linterpolation module . . . 165
2.4.8 yade.log module . . . 166
2.4.9 yade.math module . . . 168
2.4.10 yade.minieigenHP module . . . 211
2.4.11 yade.mpy module . . . 294
2.4.12 yade.pack module . . . 298
2.4.13 yade.plot module . . . 308
2.4.14 yade.polyhedra_utils module . . . 312
2.4.15 yade.post2d module . . . 314
2.4.16 yade.potential_utils module . . . 317
2.4.17 yade.qt module . . . 319
2.4.18 yade.timing module . . . 319
2.4.19 yade.utils module . . . 321
2.4.20 yade.ymport module . . . 342
2.5 Installation . . . 347
2.5.1 Supported distributions and end-of-life policy . . . 347
2.5.2 Packages . . . 347
2.5.3 Docker . . . 349
2.5.4 Source code . . . 349
2.5.5 Speed-up compilation . . . 355
2.5.6 Cloud Computing . . . 356
2.5.7 GPU Acceleration . . . 356
2.5.8 Special builds . . . 356
2.5.9 Yubuntu . . . 357
2.6 Acknowledging Yade . . . 357

3 Yade for programmers **359**
3.1 Programmer's manual . . . 359
3.1.1 Build system . . . 359
3.1.2 Development tools . . . 360
3.1.3 Debugging . . . 362
3.1.4 Regression tests . . . 368
3.1.5 Conventions . . . 370
3.1.6 Support framework . . . 375
3.1.7 Simulation framework . . . 402
3.1.8 Runtime structure . . . 408
3.1.9 Python framework . . . 409
3.1.10 Adding a new python/C++ module . . . 411
3.1.11 Maintaining compatibility . . . 413
3.2 Yade on GitLab . . . 414
3.2.1 Fast checkout (read-only) . . . 414
3.2.2 Branches on GitLab . . . 414
3.2.3 Merge requests . . . 417
3.2.4 Guidelines for pushing . . . 418

4 Theoretical background and extensions **419**
4.1 DEM formulation . . . 419
4.2 CFD-DEM coupled simulations with Yade and OpenFOAM . . . 419
4.2.1 Supported versions and examples . . . 419
4.2.2 Background . . . 420

4.2.3 Setting up a case . . . 423
4.2.4 Post-Processing . . . 424
4.2.5 Using blockMeshDict . . . 424
4.2.6 Using polyMesh . . . 425
4.3 FEM-DEM hierarchical multiscale modeling with Yade and Escript . . . 426
4.3.1 Introduction . . . 426
4.3.2 Finite element formulation . . . 426
4.3.3 Multiscale solution procedure . . . 427
4.3.4 Work on the YADE side . . . 427
4.3.5 Work on the Escript side . . . 428
4.3.6 Example tests . . . 428
4.3.7 Disclaim . . . 429
4.4 Simulating Acoustic Emissions in Yade . . . 429
4.4.1 Summary . . . 429
4.4.2 Model description . . . 429
4.4.3 Activating the algorithm within Yade . . . 430
4.4.4 Visualizing and post processing acoustic emissions . . . 432
4.4.5 Consideration of rock heterogeneity . . . 432
4.5 Using YADE 1D vertical VANS fluid resolution . . . 434
4.5.1 DEM-fluid coupling and fluid resolution in YADE . . . 434
4.5.2 Application of drag and buoyancy forces (HydroForceEngine::action) . . . 434
4.5.3 Solid phase averaging (HydroForceEngine::averageProfile) . . . 435
4.5.4 Fluid resolution\HydroForceEngine::fluidResolution . . . 436
4.6 Potential Particles and Potential Blocks . . . 437
4.6.1 Introduction . . . 437
4.6.2 Potential Particles code (PP) . . . 438
4.6.3 Potential Blocks code (PB) . . . 438
4.6.4 Engines . . . 441
4.6.5 Contact Law . . . 442
4.6.6 Shape definition of a PP and a PB . . . 442
4.6.7 Body definition of a PP and a PB . . . 444
4.6.8 Boundary Particles . . . 444
4.6.9 Visualization . . . 445
4.6.10 Axis-Aligned Bounding Box . . . 446
4.6.11 Block Generation algorithm . . . 447
4.6.12 Examples . . . 447
4.6.13 Disclaimer . . . 447
4.6.14 References . . . 447
4.7 Bayesian Calibration using GrainLearning . . . 448
4.7.1 Installation . . . 448
4.7.2 Dynamic Systems . . . 448
4.7.3 Bayesian Filtering . . . 449
4.7.4 In Yade . . . 452
4.7.5 In GrainLearning . . . 453
4.7.6 Running Bayesian calibration . . . 455
4.7.7 Setting the stopping criteria . . . 455
4.7.8 Analyzing and visualizing the results . . . 455
4.7.9 Particle-particle collision . . . 456
4.7.10 Triaxial compression . . . 456

5 Performance enhancements **459**
5.1 Accelerating Yade's FlowEngine with GPU . . . 459
5.1.1 Summary . . . 459
5.1.2 Hardware, Software, and Model Requirements . . . 459
5.1.3 Install CUDA . . . 459
5.1.4 Install OpenBlas, and Lapack . . . 460
5.1.5 Install SuiteSparse . . . 460
5.1.6 Compile Yade . . . 460

5.1.7 Controlling the GPU . . . 461
5.1.8 Performance increase . . . 461
5.2 MPI parallelization . . . 461
5.2.1 Concepts . . . 462
5.2.2 Walkthrough . . . 463
5.2.3 MPI initialization and communications . . . 465
5.2.4 Splitting . . . 471
5.2.5 Merging . . . 474
5.2.6 Hints and problems to expect . . . 475
5.2.7 Control variables . . . 475
5.2.8 Benchmark . . . 476
5.3 Using YADE with cloud computing on Amazon EC2 . . . 476
5.3.1 Summary . . . 476
5.3.2 Launching an EC2 instance . . . 476
5.3.3 Installing YADE and managing files . . . 479
5.3.4 Plotting output in the terminal . . . 480
5.3.5 Comments . . . 480
5.4 High precision calculations . . . 481
5.4.1 Installation . . . 481
5.4.2 Supported modules . . . 482
5.4.3 Double, quadruple, octuple and higher precisions . . . 483
5.4.4 Compatibility . . . 484
5.4.5 Debugging . . . 488

6 Short-courses **491**
6.1 THM short-course . . . 491
6.1.1 Installing Yade (for Windows and Mac users) . . . 491
6.1.2 Introduction to Bash and Python . . . 492
6.1.3 Day 1 - Yade Hands-on part 1 . . . 498
6.1.4 Day 1 - Yade Hands-on part 2 . . . 499
6.1.5 Day 2 - Fluids Hands-on part 1 . . . 501
6.1.6 Day 3 - Thermal Hands-on part 1 . . . 505
6.1.7 Day 3 - Thermal Hands-on part 2 . . . 509

7 Literature **513**
7.1 Yade Technical Archive . . . 513
7.1.1 About . . . 513
7.1.2 Contribute . . . 513
7.1.3 Contact . . . 513
7.1.4 Archive . . . 513
7.2 Yade publications . . . 514
7.2.1 Journal articles . . . 514
7.2.2 Conference materials and book chapters . . . 514
7.2.3 Master and PhD theses . . . 514
7.2.4 *Yade Technical Archive* . . . 514
7.3 References . . . 514

8 Yade community events **515**
8.1 Yade community events . . . 515
8.1.1 2nd Yade hackathon . . . 515
8.1.2 1st Yade hackathon . . . 515
8.1.3 2nd **Y**et **A**nother **D**iscrete **E**lement Workshop . . . 518
8.1.4 1st **Y**et **A**nother **D**iscrete **E**lement Workshop . . . 518

9 Indices and tables **521**

Bibliography **523**

Python Module Index **563**

Chapter 1

Guided tour

1.1 Introduction

1.1.1 Getting started

Before you start moving around in Yade, you should have some prior knowledge.

- Basics of command line in your Linux system are necessary for running yade. Look on the web for tutorials.
- Python language; we recommend the official Python tutorial. Reading further documents on the topic, such as Dive into Python will certainly not hurt either.

You are advised to try all commands described yourself. Don't be afraid to experiment.

Hint

Sometimes reading this documentation in a .pdf format can be more comfortable. For example in okular pdf viewer clicking links is faster than a page refresh in the web browser and to go back press the shortcut `Alt Shift ←`. To try it have a look at the inheritance graph of *PartialEngine* then go back.

Starting yade

Yade is being run primarily from terminal; the name of command is `yade`.[1] (In case you did not install from package, you might need to give specific path to the command[2]):

```
$ yade
Welcome to Yade
TCP python prompt on localhost:9001, auth cookie `sdksuy'
```

(continues on next page)

[1] The executable name can carry a suffix, such as version number (`yade-0.20`), depending on compilation options. Packaged versions on Debian systems always provide the plain `yade` alias, by default pointing to latest stable version (or latest snapshot, if no stable version is installed). You can use `update-alternatives` to change this.

[2] In general, Unix *shell* (command line) has environment variable `PATH` defined, which determines directories searched for executable files if you give name of the file without path. Typically, $PATH contains `/usr/bin/`, `/usr/local/bin`, `/bin` and others; you can inspect your `PATH` by typing `echo $PATH` in the shell (directories are separated by `:`).

If Yade executable is not in directory contained in `PATH`, you have to specify it by hand, i.e. by typing the path in front of the filename, such as in `/home/user/bin/yade` and similar. You can also navigate to the directory itself (`cd ~/bin/yade`, where `~` is replaced by your home directory automatically) and type `./yade` then (the `.` is the current directory, so `./` specifies that the file is to be found in the current directory).

To save typing, you can add the directory where Yade is installed to your `PATH`, typically by editing `~/.profile` (in normal cases automatically executed when shell starts up) file adding line like `export PATH=/home/user/bin:$PATH`. You can also define an *alias* by saying `alias yade="/home/users/bin/yade"` in that file.

Details depend on what shell you use (bash, zsh, tcsh, …) and you will find more information in introductory material on Linux/Unix.

(continued from previous page)

```
TCP info provider on localhost:21000
[[ ^L clears screen, ^U kills line. F12 controller, F11 3d view, F10 both, F9␣
↪generator, F8 plot. ]]
Yade [1]:
```

These initial lines give you some information about

- some information for *Remote control*, which you are unlikely to need now;
- basic help for the command-line that just appeared (`Yade [1]:`).

Type `quit()`, `exit()` or simply press `^D` (`^` is a commonly used written shortcut for pressing the `Ctrl` key, so here `^D` means `Ctrl D`) to quit Yade.

The command-line is ipython, python shell with enhanced interactive capabilities; it features persistent history (remembers commands from your last sessions), searching and so on. See ipython's documentation for more details.

Typically, you will not type Yade commands by hand, but use *scripts*, python programs describing and running your simulations. Let us take the most simple script that will just print "Hello world!":

```
print("Hello world!")
```

Saving such script as `hello.py`, it can be given as argument to Yade:

```
$ yade hello.py
Welcome to Yade
TCP python prompt on localhost:9001, auth cookie `askcsu'
TCP info provider on localhost:21000
Running script hello.py                                    ## the script is␣
↪being run
Hello world!                                               ## output from the␣
↪script
[[ ^L clears screen, ^U kills line. F12 controller, F11 3d view, F10 both, F9␣
↪generator, F8 plot. ]]
Yade [1]:
```

Yade will run the script and then drop to the command-line again.[3] If you want Yade to quit immediately after running the script, use the `-x` switch:

```
$ yade -x script.py
```

There is more command-line options than just `-x`, run `yade -h` to see all of them.

Options:

-v, --version show program's version number and exit

-h, --help show this help message and exit

-j THREADS, --threads=THREADS Number of OpenMP threads to run; defaults to 1. Equivalent to setting OMP_NUM_THREADS environment variable.

--cores=CORES Set number of OpenMP threads (as --threads) and in addition set affinity of threads to the cores given.

--update Update deprecated class names in given script(s) using text search & replace. Changed files will be backed up

[3] Plain Python interpreter exits once it finishes running the script. The reason why Yade does the contrary is that most of the time script only sets up simulation and lets it run; since computation typically runs in background thread, the script is technically finished, but the computation is running.

with ~ suffix. Exit when done without running any simulation.

--nice=NICE Increase nice level (i.e. decrease priority) by given number.

-x Exit when the script finishes

-f Set *logging verbosity*, default is -f3 (yade.log.WARN) for all classes

-n Run without graphical interface (equivalent to unsetting the DISPLAY environment variable)

--test Run regression test suite and exit; the exists status is 0 if all tests pass, 1 if a test fails and 2 for an unspecified exception.

--check Run a series of user-defined check tests as described in scripts/checks-and-tests/checks/README and *Regression tests*

--performance Starts a test to measure the productivity.

--stdperformance Starts a standardized test to measure the productivity, which will keep retrying to run the benchmark until standard deviation of the performance is below 1%. A common type of simulation is done: the spheres fall down in a box and are given enough time to settle in there. Note: better to use this with argument *-j THREADS* (explained above).

--quickperformance Starts a quick test to measure the productivity. Same as above, but only two short runs are performed, without the attempts to find the computer performance with small error.

--no-gdb Do not show backtrace when yade crashes (only effective with --debug)[4].

Quick inline help

All of functions callable from ipython shell have a quickly accessible help by appending `?` to the function name, or calling `help(…)` command on them:

```
Yade [1]: O.run?
[31mDocstring:[39m
run( (Omega)arg1 [, (int)nSteps=-1 [, (bool)wait=False]]) -> None :
    Run the simulation. *nSteps* how many steps to run, then stop (if positive);␣
↪*wait* will cause not returning to python until simulation will have stopped.
[31mType:[39m      method

Yade [2]: help(O.pause)
Help on method pause in module yade.wrapper:

pause(...) method of yade.wrapper.Omega instance
    pause( (Omega)arg1) -> None :
        Stop simulation execution. (May be called from within the loop, and it will␣
↪stop after the current step).
```

A quick way to discover available functions is by using the tab-completion mechanism, e.g. type `O.` then press tab.

[4] On some linux systems stack trace will produce `Operation not permitted` error. See *debugging section* for solution.

Creating simulation

To create simulation, one can either use a specialized class of type *FileGenerator* to create full scene, possibly receiving some parameters. Generators are written in C++ and their role is limited to well-defined scenarios. For instance, to create triaxial test scene:

```
Yade [3]: TriaxialTest(numberOfGrains=200).load()

Yade [4]: len(O.bodies)
Out[4]: 206
```

Generators are regular yade objects that support attribute access.

It is also possible to construct the scene by a python script; this gives much more flexibility and speed of development and is the recommended way to create simulation. Yade provides modules for streamlined body construction, import of geometries from files and reuse of common code. Since this topic is more involved, it is explained in the *User's manual*.

Running simulation

As explained below, the loop consists in running defined sequence of engines. Step number can be queried by `O.iter` and advancing by one step is done by `O.step()`. Every step advances *virtual time* by current timestep, `O.dt` that can be directly assigned or, which is usually better, automatically determined by a *GlobalStiffnessTimeStepper*, if present:

```
Yade [5]: O.iter
Out[5]: 0

Yade [6]: O.time
Out[6]: 0.0

Yade [7]: O.dt=1e-4

Yade [8]: O.dynDt=False #else it would be adjusted automaticaly during first iteration

Yade [9]: O.step()

Yade [10]: O.iter
Out[10]: 1

Yade [11]: O.time
Out[11]: 0.0001
```

Normal simulations, however, are run continuously. Starting/stopping the loop is done by `O.run()` and `O.pause()`; note that `O.run()` returns control to Python and the simulation runs in background; if you want to wait for it to finish, use `O.wait()`. Fixed number of steps can be run with `O.run(1000)`, `O.run(1000,True)` will run and wait. To stop at absolute step number, `O.stopAtIter` can be set and `O.run()` called normally.

```
Yade [12]: O.run()

Yade [13]: O.pause()

Yade [14]: O.iter
Out[14]: 1178

Yade [15]: O.run(100000,True)

Yade [16]: O.iter
```

(continues on next page)

(continued from previous page)

```
Out[16]: 101178

Yade [17]: O.stopAtIter=500000

Yade [18]: O.run()

Yade [19]: O.wait()

Yade [20]: O.iter
Out[20]: 500000
```

Saving and loading

Simulation can be saved at any point to a binary file (optionaly compressed if the filename has extensions such as ".gz" or ".bz2"). Saving to a XML file is also possible though resulting in larger files and slower save/load, it is used when the filename contains "xml". With some limitations, it is generally possible to load the scene later and resume the simulation as if it were not interrupted. Note that since the saved scene is a dump of Yade's internal objects, it might not (probably will not) open with different Yade version. This problem can be sometimes solved by migrating the saved file using ".xml" format.

```
Yade [21]: O.save('/tmp/a.yade.bz2')

Yade [22]: O.reload()

Yade [23]: O.load('/tmp/another.yade.bz2')
```

The principal use of saving the simulation to XML is to use it as temporary in-memory storage for checkpoints in simulation, e.g. for reloading the initial state and running again with different parameters (think tension/compression test, where each begins from the same virgin state). The functions `O.saveTmp()` and `O.loadTmp()` can be optionally given a slot name, under which they will be found in memory:

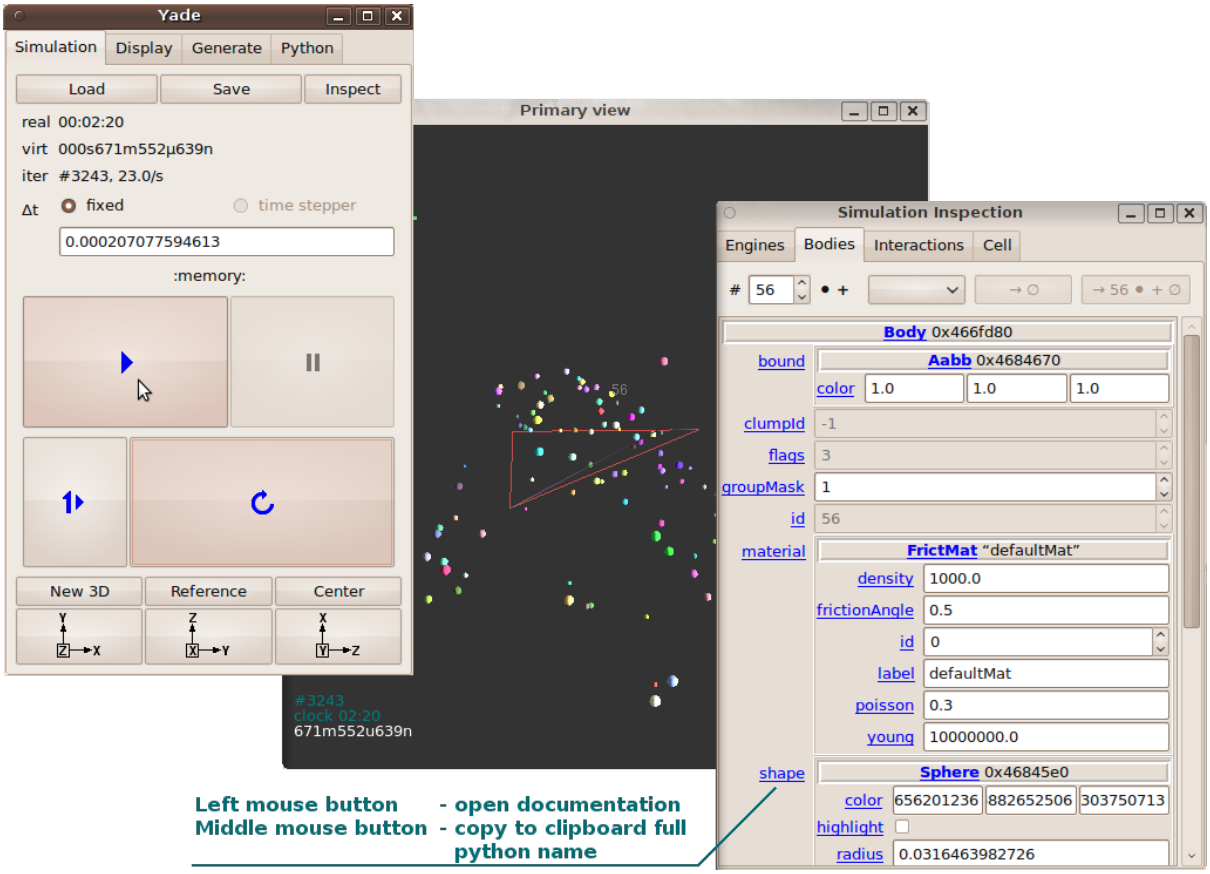
```
Yade [24]: O.saveTmp()

Yade [25]: O.loadTmp()

Yade [26]: O.saveTmp('init') ## named memory slot

Yade [27]: O.loadTmp('init')
```

Simulation can be reset to empty state by `O.reset()`.

It can be sometimes useful to run different simulation, while the original one is temporarily suspended, e.g. when dynamically creating packing. `O.switchWorld()` toggles between the primary and secondary simulation.

Graphical interface

Yade can be optionally compiled with QT based graphical interface (qt4 and qt5 are supported). It can be started by pressing `F12` in the command-line, and also is started automatically when running a script.



The control window on the left (fig. *imgQtGui*) is called `Controller` (can be invoked by `yade.qt.Controller()` from python or by pressing `F12` key in terminal):

1. The *Simulation* tab is mostly self-explanatory, and permits basic simulation control.
2. The *Display* tab has various rendering-related options, which apply to all opened views (they can be zero or more, new one is opened by the *New 3D* button).
3. The *Python* tab has only a simple text entry area; it can be useful to enter python commands while the command-line is blocked by running script, for instance.

Inside the *Inspect* window (on the right in fig. *imgQtGui*) all simulation data can be examined and modified in realtime.

1. Clicking left mouse button on any of the blue hyperlinks will open documentation.
2. Clicking middle mouse button will copy the fully qualified python name into clipboard, which can be pasted into terminal by clicking middle mouse button in the terminal (or pressing `Ctrl-V`).

3d views can be controlled using mouse and keyboard shortcuts; help is displayed if you press the `h` key while in the 3d view. Note that having the 3d view open can slow down running simulation significantly, it is meant only for quickly checking whether the simulation runs smoothly. Advanced post-processing is described in dedicated section *Data mining*.

1.1.2 Architecture overview

In the following, a high-level overview of Yade architecture will be given. As many of the features are directly represented in simulation scripts, which are written in Python, being familiar with this language will help you follow the examples. For the rest, this knowledge is not strictly necessary and you can ignore code examples.

Data and functions

To assure flexibility of software design, yade makes clear distinction of 2 families of classes: *data* components and *functional* components. The former only store data without providing functionality, while the latter define functions operating on the data. In programming, this is known as *visitor* pattern (as functional components "visit" the data, without being bound to them explicitly).

Entire simulation, i.e. both data and functions, are stored in a single `Scene` object. It is accessible through the *Omega* class in python (a singleton), which is by default stored in the `O` global variable:

```
Yade [28]: O.bodies        # some data components
Out[28]: <yade.wrapper.BodyContainer at 0x7f97f1992260>

Yade [29]: len(O.bodies)  # there are no bodies as of yet
Out[29]: 0

Yade [30]: O.engines       # functional components, empty at the moment
Out[30]: []
```

Data components

Bodies

Yade simulation (class *Scene*, but hidden inside *Omega* in Python) is represented by *Bodies*, their *Interactions* and resultant generalized *forces* (all stored internally in special containers).

Each *Body* comprises the following:

Shape
: represents particle's geometry (neutral with regards to its spatial orientation), such as *Sphere*, *Facet* or inifinite *Wall*; it usually does not change during simulation.

Material
: stores characteristics pertaining to mechanical behavior, such as Young's modulus or density, which are independent on particle's shape and dimensions; usually constant, might be shared amongst multiple bodies.

State
: contains state variables, in particular spatial *position* and *orientation*, *linear* and *angular* velocity; it is updated by the *integrator* at every step. The derived classes would contain other information related to current state of this body, e.g. its temperature, *averaged damage* or *broken links* between components.

Bound
: is used for approximate ("pass 1") contact detection; updated as necessary following body's motion. Currently, *Aabb* is used most often as *Bound*. Some bodies may have no *Bound*, in which case they are exempt from contact detection.

(In addition to these 4 components, bodies have several more minor data associated, such as *Body::id* or *Body::mask*.)

All these four properties can be of different types, derived from their respective base types. Yade frequently makes decisions about computation based on those types: *Sphere* + *Sphere* collision has to be treated differently than *Facet* + *Sphere* collision. Objects making those decisions are called *Dispatchers* and are essential to understand Yade's functioning; they are discussed below.

Explicitly assigning all 4 properties to each particle by hand would be not practical; there are utility functions defined to create them with all necessary ingredients. For example, we can create sphere particle using *utils.sphere*:

```
Yade [31]: s=utils.sphere(center=[0,0,0],radius=1)
```

(continues on next page)

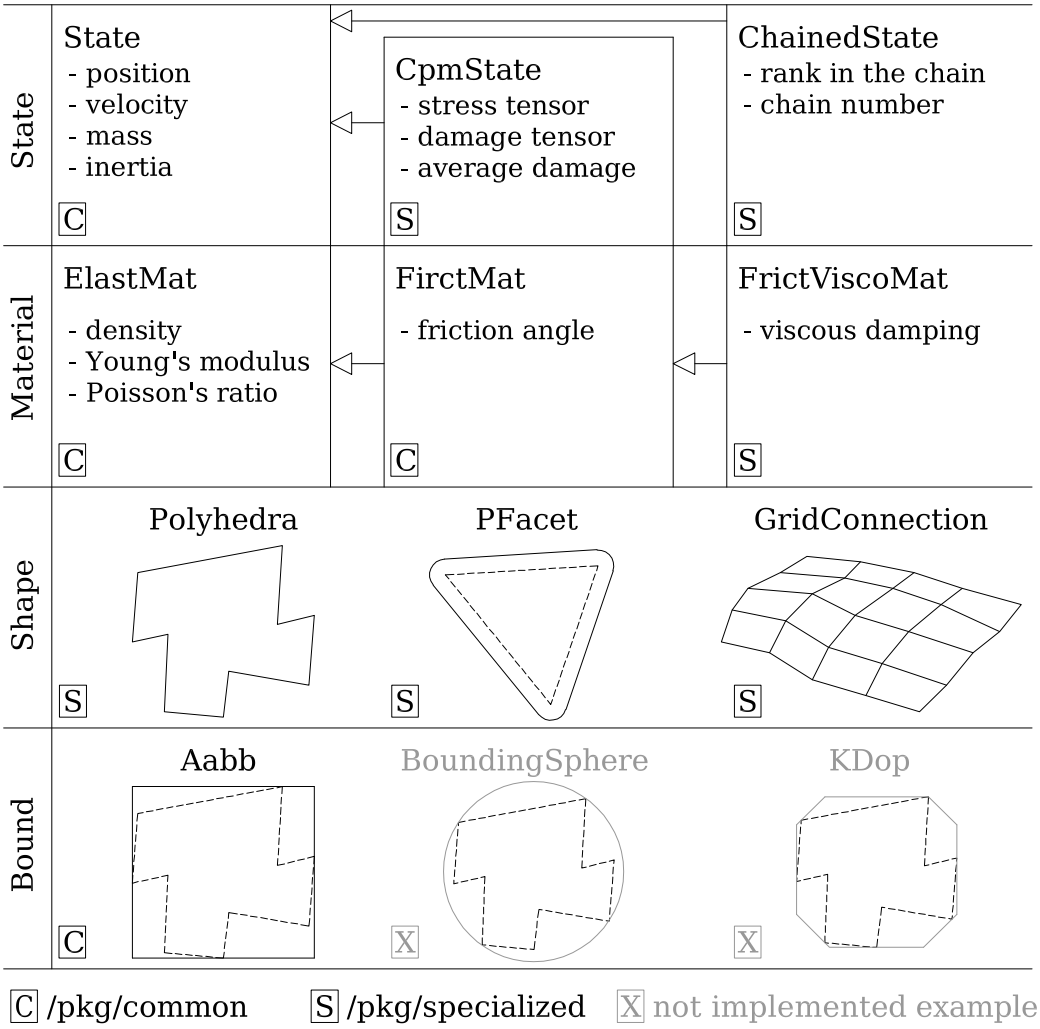


Fig. 1: Examples of concrete classes that might be used to describe a *Body*: *State*, *CpmState*, *ChainedState*, *Material*, *ElastMat*, *FrictMat*, *FrictViscoMat*, *Shape*, *Polyhedra*, *PFacet*, *GridConnection*, *Bound*, *Aabb*.

(continued from previous page)

```
Yade [32]: s.shape, s.state, s.mat, s.bound
Out[32]:
(<Sphere instance at 0x49260a0>,
 <State instance at 0x492d390>,
 <FrictMat instance at 0x48decc0>,
 None)

Yade [33]: s.state.pos
Out[33]: Vector3(0,0,0)

Yade [34]: s.shape.radius
Out[34]: 1.0
```

We see that a sphere with material of type *FrictMat* (default, unless you provide another *Material*) and bounding volume of type *Aabb* (axis-aligned bounding box) was created. Its position is at the origin and its radius is 1.0. Finally, this object can be inserted into the simulation; and we can insert yet one sphere as well.

```
Yade [35]: O.bodies.append(s)
Out[35]: 0

Yade [36]: O.bodies.append(utils.sphere([0,0,2],.5))
Out[36]: 1
```

In each case, return value is *Body.id* of the body inserted.

Since till now the simulation was empty, its id is 0 for the first sphere and 1 for the second one. Saving the id value is not necessary, unless you want to access this particular body later; it is remembered internally in *Body* itself. You can address bodies by their id:

```
Yade [37]: O.bodies[1].state.pos
Out[37]: Vector3(0,0,2)

Yade [38]: O.bodies[100]    # error because there are only two bodies
[31m---------------------------------------------------------------------------[39m
[31mIndexError[39m                                Traceback (most recent call last)
[36mCell[39m[36m [39m[32mIn[38][39m[32m, line 1[39m
[32m----> [39m[32m1[39m␣
↪[43mO[49m[43m.[49m[43mbodies[49m[43m[[49m[32;43m100[39;49m[43m][49m    ␣
↪[38;5;66;03m# error because there are only two bodies[39;00m

[31mIndexError[39m: Body id out of range.
```

Adding the same body twice is, for reasons of the id uniqueness, not allowed:

```
Yade [39]: O.bodies.append(s)  # error because this sphere was already added
[31m---------------------------------------------------------------------------[39m
[31mIndexError[39m                                Traceback (most recent call last)
[36mCell[39m[36m [39m[32mIn[39][39m[32m, line 1[39m
[32m----> [39m[32m1[39m␣
↪[43mO[49m[43m.[49m[43mbodies[49m[43m.[49m[43mappend[49m[43m([49m[43ms[49m[43m)[49m ␣
↪[38;5;66;03m# error because this sphere was already added[39;00m

[31mIndexError[39m: Body already has id 0 set; appending such body (for the second␣
↪time) is not allowed.
```

Bodies can be iterated over using standard python iteration syntax:

```
Yade [40]: for b in O.bodies:
   ....:    print(b.id,b.shape.radius)
   ....:
0 1.0
1 0.5
```

Interactions

Interactions are always between pair of bodies; usually, they are created by the collider based on spatial proximity; they can, however, be created explicitly and exist independently of distance. Each interaction has 2 components:

IGeom
: holding geometrical configuration of the two particles in collision; it is updated automatically as the particles in question move and can be queried for various geometrical characteristics, such as penetration distance or shear strain.

  Based on combination of types of *Shapes* of the particles, there might be different storage requirements; for that reason, a number of derived classes exists, e.g. for representing geometry of contact between *Sphere+Sphere*, *Cylinder+Sphere* etc. Note, however, that it is possible to represent many type of contacts with the basic sphere-sphere geometry (for instance in *Ig2_Wall_Sphere_ScGeom*).

IPhys
: representing non-geometrical features of the interaction; some are computed from *Materials* of the particles in contact using some averaging algorithm (such as contact stiffness from Young's moduli of particles), others might be internal variables like damage.

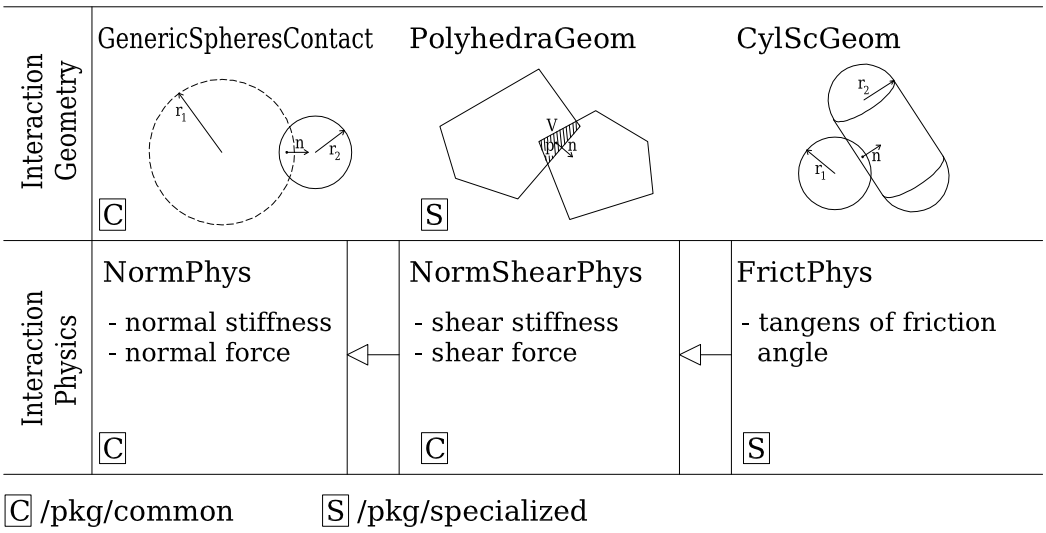


Fig. 2: Examples of concrete classes that might be used to describe an *Interaction*: *IGeom*, *GenericSpheresContact*, *PolyhedraGeom*, *CylScGeom*, *IPhys*, *NormPhys*, *NormShearPhys*, *FrictPhys*.

Suppose now interactions have been already created. We can access them by the id pair:

```
Yade [41]: O.interactions[0,1]
Out[41]: <Interaction instance at 0x48b18f0>

Yade [42]: O.interactions[1,0]     # order of ids is not important
Out[42]: <Interaction instance at 0x48b18f0>

Yade [43]: i=O.interactions[0,1]
```

(continues on next page)

(continued from previous page)

```
Yade [44]: i.id1,i.id2
Out[44]: (0, 1)

Yade [45]: i.geom
Out[45]: <ScGeom instance at 0x49177e0>

Yade [46]: i.phys
Out[46]: <FrictPhys instance at 0x4a12cd0>

Yade [47]: O.interactions[100,10111]     # asking for non existing interaction throws␣
↪exception
[31m---------------------------------------------------------------------------[39m
[31mIndexError[39m                                Traceback (most recent call last)
[36mCell[39m[36m [39m[32mIn[47][39m[32m, line 1[39m
[32m----> [39m[32m1[39m␣
↪[43mO[49m[43m.[49m[43minteractions[49m[43m[[49m[32;43m100[39;49m[43m,[49m[32;43m10111[39;49m[43m][
↪    [38;5;66;03m# asking for non existing interaction throws exception[39;00m

[31mIndexError[39m: No such interaction
```

Generalized forces

Generalized forces include force, torque and forced displacement and rotation; they are stored only temporariliy, during one computation step, and reset to zero afterwards. For reasons of parallel computation, they work as accumulators, i.e. only can be added to, read and reset.

```
Yade [48]: O.forces.f(0)
Out[48]: Vector3(0,0,0)

Yade [49]: O.forces.addF(0,Vector3(1,2,3))

Yade [50]: O.forces.f(0)
Out[50]: Vector3(1,2,3)
```

You will only rarely modify forces from Python; it is usually done in c++ code and relevant documentation can be found in the Programmer's manual.

Function components

In a typical DEM simulation, the following sequence is run repeatedly:

- reset forces on bodies from previous step
- approximate collision detection (pass 1)
- detect exact collisions of bodies, update interactions as necessary
- solve interactions, applying forces on bodies
- apply other external conditions (gravity, for instance).
- change position of bodies based on forces, by integrating motion equations.

Each of these actions is represented by an *Engine*, functional element of simulation. The sequence of engines is called *simulation loop*.

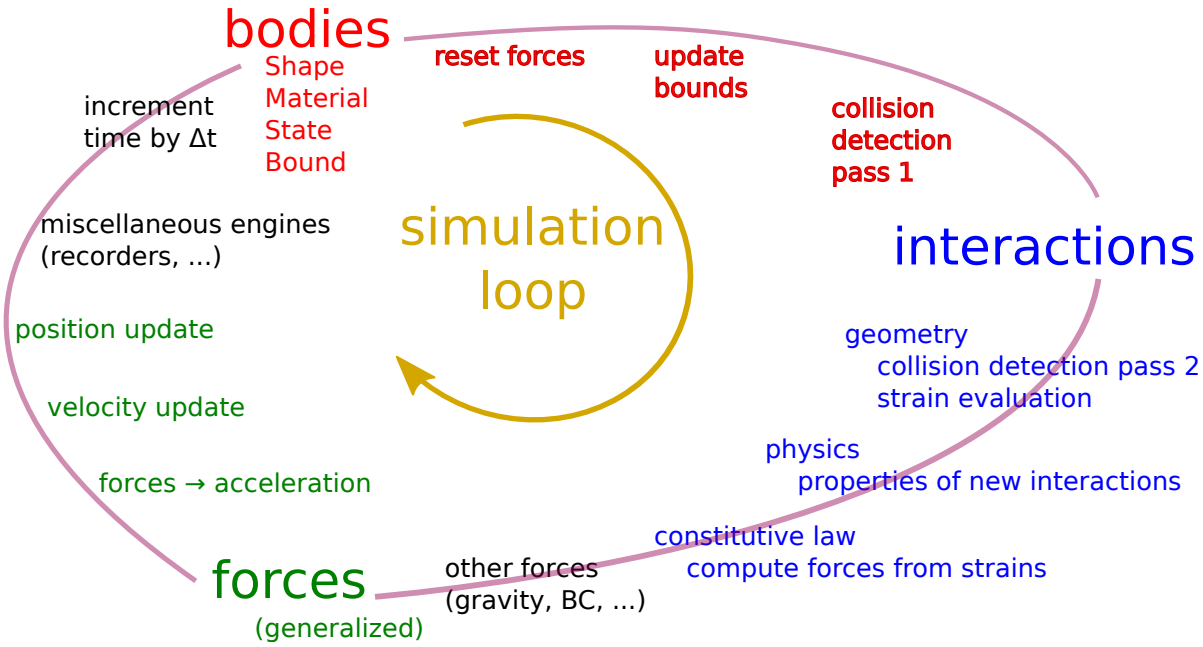


Fig. 3: Typical simulation loop; each step begins at body-centered bit at 11 o'clock, continues with interaction bit, force application bit, miscellanea and ends with time update.

Engines

Simulation loop, shown at fig. *img-yade-iter-loop*, can be described as follows in Python (details will be explained later); each of the `O.engines` items is instance of a type deriving from *Engine*:

```
O.engines=[
        # reset forces
        ForceResetter(),
        # approximate collision detection, create interactions
        InsertionSortCollider([Bo1_Sphere_Aabb(),Bo1_Facet_Aabb()]),
        # handle interactions
        InteractionLoop(
                [Ig2_Sphere_Sphere_ScGeom(),Ig2_Facet_Sphere_ScGeom()],
                [Ip2_FrictMat_FrictMat_FrictPhys()],
                [Law2_ScGeom_FrictPhys_CundallStrack()],
        ),
        # apply other conditions
        GravityEngine(gravity=(0,0,-9.81)),
        # update positions using Newton's equations
        NewtonIntegrator()
]
```

There are 3 fundamental types of Engines:

GlobalEngines
: operating on the whole simulation (e.g. *ForceResetter* which zeroes forces acting on bodies or *GravityEngine* looping over all bodies and applying force based on their mass)

PartialEngine
: operating only on some pre-selected bodies (e.g. *ForceEngine* applying constant force to some *selected* bodies)

Dispatchers
: do not perform any computation themselves; they merely call other functions, represented by function objects, *Functors*. Each functor is specialized, able to handle certain object types, and

will be dispatched if such obejct is treated by the dispatcher.

Dispatchers and functors

For approximate collision detection (pass 1), we want to compute *bounds* for all *bodies* in the simulation; suppose we want bound of type *axis-aligned bounding box*. Since the exact algorithm is different depending on particular *shape*, we need to provide functors for handling all specific cases. In the `O.engines=[…]` declared above, the line:

```
InsertionSortCollider([Bo1_Sphere_Aabb(),Bo1_Facet_Aabb()])
```

creates *InsertionSortCollider* (it internally uses *BoundDispatcher*, but that is a detail). It traverses all bodies and will, based on *shape* type of each *body*, dispatch one of the functors to create/update *bound* for that particular body. In the case shown, it has 2 functors, one handling *spheres*, another *facets*.

The name is composed from several parts: `Bo` (functor creating *Bound*), which accepts `1` type *Sphere* and creates an *Aabb* (axis-aligned bounding box; it is derived from *Bound*). The *Aabb* objects are used by *InsertionSortCollider* itself. All `Bo1` functors derive from *BoundFunctor*.

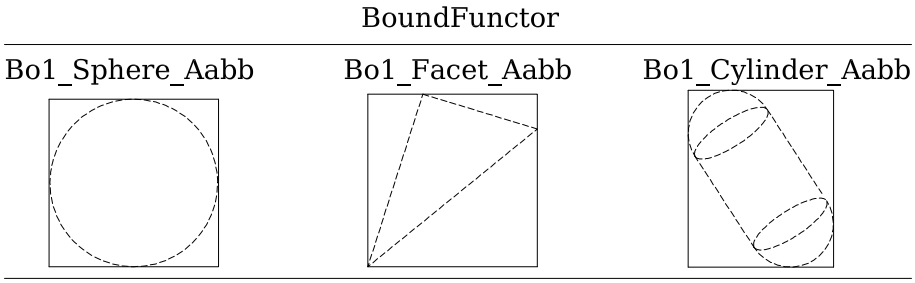


Fig. 4: Example *bound functors* producing *Aabb* accepting various different types, such as *Sphere*, *Facet* or *Cylinder*. In the case shown, the `Bo1` functors produce *Aabb* instances from single specific *Shape*, hence the number `1` in the functor name. Each of those functors uses specific geometry of the *Shape* i.e. position of nodes in *Facet* or *radius of sphere* to calculate the *Aabb*.

The next part, reading

```
InteractionLoop(
        [Ig2_Sphere_Sphere_ScGeom(),Ig2_Facet_Sphere_ScGeom()],
        [Ip2_FrictMat_FrictMat_FrictPhys()],
        [Law2_ScGeom_FrictPhys_CundallStrack()],
),
```

hides 3 internal dispatchers within the *InteractionLoop* engine; they all operate on interactions and are, for performance reasons, put together:

IGeomDispatcher* which uses *IGeomFunctor

uses the first set of functors (`Ig2`), which are dispatched based on combination of `2` *Shapes* objects. Dispatched functor resolves exact collision configuration and creates an Interaction Geometry *IGeom* (whence `Ig` in the name) associated with the interaction, if there is collision. The functor might as well determine that there is no real collision even if they did overlap in the approximate collision detection (e.g. the *Aabb* did overlap, but the shapes did not). In that case the attribute is set to false and interaction is scheduled for removal.

1. The first functor, *Ig2_Sphere_Sphere_ScGeom*, is called on interaction of 2 *Spheres* and creates *ScGeom* instance, if appropriate.
2. The second functor, *Ig2_Facet_Sphere_ScGeom*, is called for interaction of *Facet* with *Sphere* and might create (again) a *ScGeom* instance.

All `Ig2` functors derive from *IGeomFunctor* (they are documented at the same place).

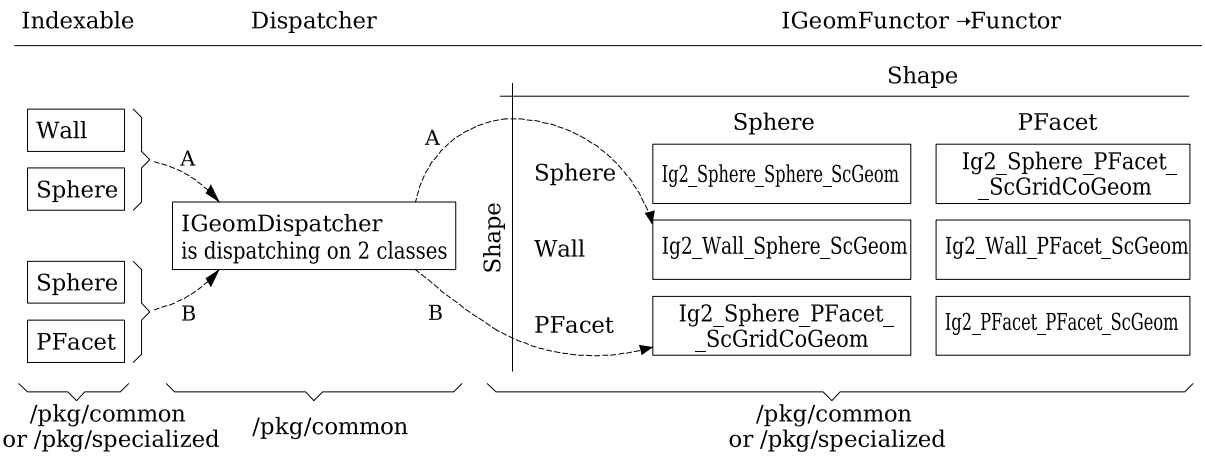


Fig. 5: Example *interaction geometry functors* producing *ScGeom* or *ScGridCoGeom* accepting two various different types (hence `2` in their name `Ig2`), such as *Sphere*, *Wall* or *PFacet*. Each of those functors uses specific geometry of the *Shape* i.e. position of nodes in *PFacet* or *radius of sphere* to calculate the *interaction geometry*.

IPhysDispatcher* which uses *IPhysFunctor

dispatches to the second set of functors based on combination of `2` *Materials*; these functors return return *IPhys* instance (the `Ip` prefix). In our case, there is only 1 functor used, *Ip2_FrictMat_FrictMat_FrictPhys*, which create *FrictPhys* from 2 *FrictMat's*.

`Ip2` functors are derived from *IPhysFunctor*.

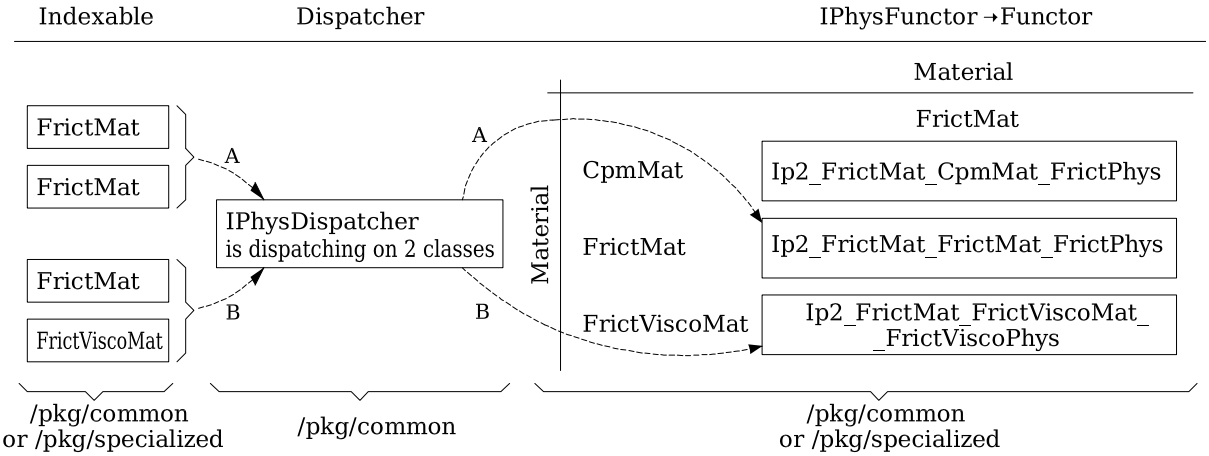


Fig. 6: Example *interaction physics functors* (*Ip2_FrictMat_CpmMat_FrictPhys*, *Ip2_FrictMat_FrictMat_FrictPhys* and *Ip2_FrictMat_FrictViscoMat_FrictViscoPhys*) producing *FrictPhys* or *FrictViscoPhys* accepting two various different types of *Material* (hence `Ip2`), such as *CpmMat*, *FrictMat* or *FrictViscoMat*.

LawDispatcher* which uses *LawFunctor

dispatches to the third set of functors, based on combinations of *IGeom* and *IPhys* (wherefore `2` in their name again) of each particular interaction, created by preceding functors. The `Law2` functors represent constitutive law; they resolve the interaction by computing forces on the interacting bodies (repulsion, attraction, shear forces, …) or otherwise update interaction state variables.

`Law2` functors all inherit from *LawFunctor*.

There is chain of types produced by earlier functors and accepted by later ones; the user is responsible to satisfy type requirement (see img. *img-dispatch-loop*). An exception (with explanation) is raised in the contrary case.

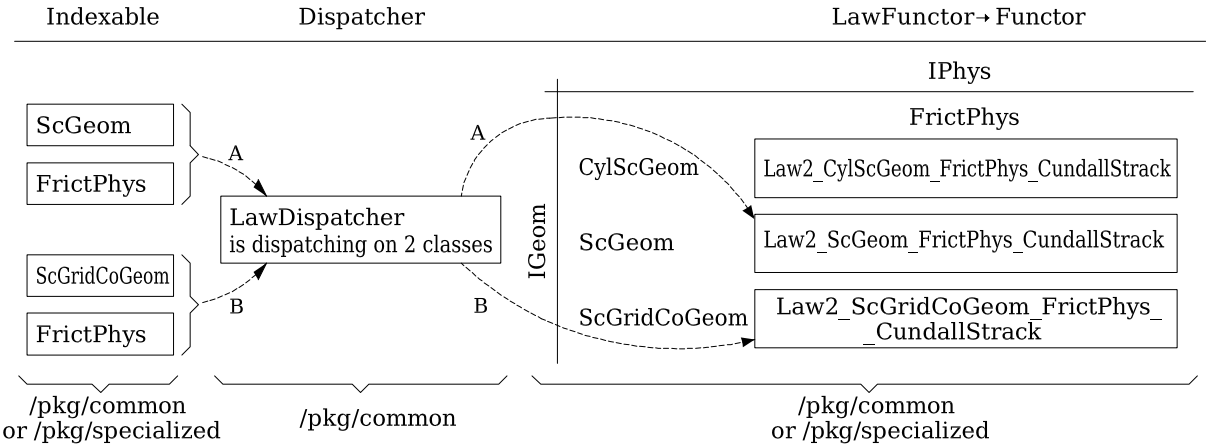


Fig. 7: Example *LawFunctors* (*Law2_CylScGeom_FrictPhys_CundallStrack*, *Law2_ScGeom_FrictPhys_CundallStrack* and *Law2_ScGridCoGeom_FrictPhys_CundallStrack*) each of them performing calcuation of forces according to selected constitutive law.

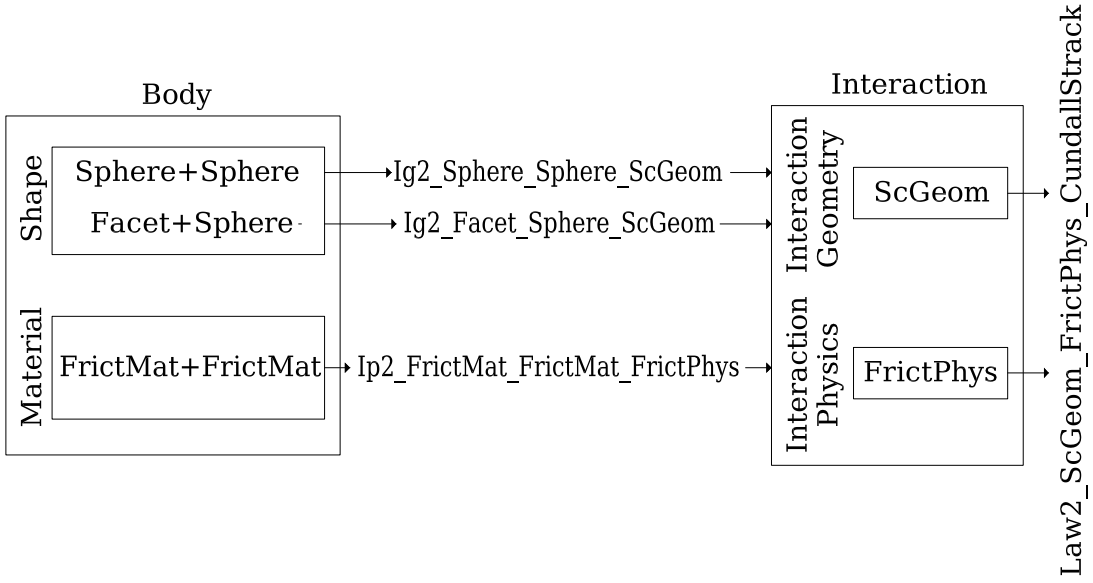


Fig. 8: Chain of functors producing and accepting certain types. In the case shown, the `Ig2` functors produce *ScGeom* instances from all handled *Shapes* combinations; the `Ig2` functor produces *FrictMat*. The constitutive law functor `Law2` accepts the combination of types produced. Note that the types are stated in the functor's class names.

Note

When Yade starts, O.engines is filled with a reasonable default list, so that it is not strictly necessary to redefine it when trying simple things. The default scene will handle spheres, boxes, and facets with *frictional* properties correctly, and adjusts the timestep dynamically. You can find an example in examples/simple-scene/simple-scene-default-engines.py.

1.2 Tutorial

This tutorial originated as handout for a course held at Technische Universität Dresden / Fakultät Bauingenieurwesen / Institut für Geotechnik in Jaunary 2011. The focus was to give quick and rather practical introduction to people without prior modeling experience, but with knowledge of mechanics. Some computer literacy was assumed, though basics are reviewed in the *Hands-on section*.

The course did not in reality follow this document, but was based on interactive writing and commenting simple *Examples*, which were mostly suggested by participants; many thanks to them for their ideas and suggestions.

1.2.1 Introduction

The chapter *Introduction* is summarized in following presentation Yade: past, present and future with some additional different examples. This presentation is from year 2011 and does not include latest additions. As of year 2019 it is factually correct.

1.2.2 Hands-on

Shell basics

Directory tree

Directory tree is hierarchical way to organize files in operating systems. A typical (reduced) tree in linux looks like this:

```
/            Root
├── boot         System startup
├── bin          Low-level programs
├── lib          Low-level libraries
├── dev          Hardware access
├── sbin         Administration programs
├── proc         System information
├── var          Files modified by system services
├── root         Root (administrator) home directory
├── etc          Configuration files
├── media        External drives
├── tmp          Temporary files
├── usr          Everything for normal operation (usr = UNIX system resources)
│   ├── bin        User programs
│   ├── sbin       Administration programs
│   ├── include    Header files for c/c++
│   ├── lib        Libraries
│   ├── local      Locally installed software
│   └── doc        Documentation
└── home         Contains the user's home directories
    ├── user       Home directory for user
    └── user1      Home directory for user1
```

Note that there is a single root `/`; all other disks (such as USB sticks) attach to some point in the tree (e.g. in `/media`).

Shell navigation

Shell is the UNIX command-line, interface for conversation with the machine. Don't be afraid.

Moving around

The shell is always operated by some `user`, at some concrete `machine`; these two are constant. We can move in the directory structure, and the current place where we are is *current directory*. By default, it is the *home directory* which contains all files belonging to the respective user:

```
user@machine:~$                          # user operating at machine, in the directory␣
↪~ (= user's home directory)
user@machine:~$ ls .                     # list contents of the current directory
user@machine:~$ ls foo                   # list contents of directory foo, relative to␣
↪the dcurrent directory ~ (= ls ~/foo = ls /home/user/foo)
user@machine:~$ ls /tmp                  # list contents of /tmp
user@machine:~$ cd foo                   # change directory to foo
user@machine:~/foo$ ls ~                 # list home directory (= ls /home/user)
user@machine:~/foo$ cd bar               # change to bar (= cd ~/foo/bar)
user@machine:~/foo/bar$ cd ../../foo2    # go to the parent directory twice, then to␣
↪foo2 (cd ~/foo/bar/../../foo2 = cd ~/foo2 = cd /home/user/foo2)
user@machine:~/foo2$ cd                  # go to the home directory (= ls ~ = ls /
↪home/user)
user@machine:~$
```

Users typically have only permissions to write (i.e. modify files) only in their home directory (abbreviated `~`, usually is `/home/user`) and `/tmp`, and permissions to read files in most other parts of the system:

```
user@machine:~$ ls /root    # see what files the administrator has
ls: cannot open directory /root: Permission denied
```

Keys

Useful keys on the command-line are:

<tab>	show possible completions of what is being typed (use abundantly!)
^C (=Ctrl+C)	delete current line
^D	exit the shell
↑↓	move up and down in the command history
^C	interrupt currently running program
^\	kill currently running program
Shift-PgUp	scroll the screen up (show past output)
Shift-PgDown	scroll the screen down (show future output; works only on quantum computers)

Running programs

When a program is being run (without giving its full path), several directories are searched for program of that name; those directories are given by `$PATH`:

```
user@machine:~$ echo $PATH     # show the value of $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
user@machine:~$ which ls       # say what is the real path of ls
```

The first part of the command-line is the program to be run (`which`), the remaining parts are *arguments* (`ls` in this case). It is up to the program which arguments it understands. Many programs can take special arguments called *options* starting with `-` (followed by a single letter) or `--` (followed by words); one of the common options is `-h` or `--help`, which displays how to use the program (try `ls --help`).

Full documentation for each program usually exists as *manual page* (or *man page*), which can be shown using e.g. `man ls` (`q` to exit)

Starting yade

If yade is installed on the machine, it can be (roughly speaking) run as any other program; without any arguments, it runs in the "dialog mode", where a command-line is presented:

```
user@machine:~$ yade
Welcome to Yade 2019.01a
TCP python prompt on localhost:9002, auth cookie `adcusk'
XMLRPC info provider on http://localhost:21002
[[ ^L clears screen, ^U kills line. F12 controller, F11 3d view, F10 both, F9␣
 ↪generator, F8 plot. ]]
Yade [1]:                                              #### hit ^D to exit
Do you really want to exit ([y]/n)?
Yade: normal exit.
```

The command-line is in fact `python`, enriched with some yade-specific features. (Pure python interpreter can be run with `python` or `ipython` commands).

Instead of typing commands on-by-one on the command line, they can be be written in a file (with the .py extension) and given as argument to Yade:

```
user@machine:~$ yade simulation.py
```

For a complete help, see `man yade`

Exercises

1. Open the terminal, navigate to your home directory
2. Create a new empty file and save it in `~/first.py`
3. Change directory to `/tmp`; delete the file `~/first.py`
4. Run program `xeyes`
5. Look at the help of Yade.
6. Look at the *manual page* of Yade
7. Run Yade, exit and run it again.

Python basics

We assume the reader is familar with Python tutorial and only briefly review some of the basic capabilities. The following will run in pure-python interpreter (`python` or `ipython`), but also inside Yade, which is a super-set of Python.

Numerical operations and modules:

```
Yade [1]: (1+3*4)**2          # usual rules for operator precedence, ** is␣
 ↪exponentiation
Out[1]: 169

Yade [2]: import math         # gain access to "module" of functions

Yade [3]: math.sqrt(2)        # use a function from that module
Out[3]: 1.4142135623730951

Yade [4]: import math as m    # use the module under a different name
```

(continues on next page)

(continued from previous page)

```
Yade [5]: m.cos(m.pi)
Out[5]: -1.0

Yade [6]: from math import *  # import everything so that it can be used without␣
↪module name

Yade [7]: cos(pi)
Out[7]: -1.0
```

Variables:

```
Yade [8]: a=1; b,c=2,3       # multiple commands separated with ;, multiple assignment

Yade [9]: a+b+c
Out[9]: 6
```

Sequences

Lists

Lists are variable-length sequences, which can be modified; they are written with braces `[...]`, and their elements are accessed with numerical indices:

```
Yade [10]: a=[1,2,3]          # list of numbers

Yade [11]: a[0]               # first element has index 0
Out[11]: 1

Yade [12]: a[-1]              # negative counts from the end
Out[12]: 3

Yade [13]: a[3]               # error
[31m---------------------------------------------------------------------------[39m
[31mIndexError[39m                                Traceback (most recent call last)
[36mCell[39m[36m [39m[32mIn[13][39m[32m, line 1[39m
[32m----> [39m[32m1[39m [43ma[49m[43m[[49m[32;43m3[39;49m[43m][49m
↪[38;5;66;03m# error[39;00m

[31mIndexError[39m: list index out of range

Yade [14]: len(a)             # number of elements
Out[14]: 3

Yade [15]: a[1:]              # from second element to the end
Out[15]: [2, 3]

Yade [16]: a+=[4,5]           # extend the list

Yade [17]: a+=[6]; a.append(7) # extend with single value, both have the same effect

Yade [18]: 9 in a             # test presence of an element
Out[18]: False
```

Lists can be created in various ways:

```
Yade [19]: range(10)
Out[19]: range(0, 10)

Yade [20]: range(10)[-1]
Out[20]: 9
```

List of squares of even number smaller than 20, i.e. $\left\{a^2 \ \forall a \in \{0,\cdots,19\} \ \middle| \ 2 \| a\right\}$ (note the similarity):

```
Yade [21]: [a**2 for a in range(20) if a%2==0]
Out[21]: [0, 4, 16, 36, 64, 100, 144, 196, 256, 324]
```

Tuples

Tuples are constant sequences:

```
Yade [22]: b=(1,2,3)

Yade [23]: b[0]
Out[23]: 1

Yade [24]: b[0]=4              # error
[31m---------------------------------------------------------------------------[39m
[31mTypeError[39m                                 Traceback (most recent call last)
[36mCell[39m[36m [39m[32mIn[24][39m[32m, line 1[39m
[32m----> [39m[32m1[39m [43mb[49m[43m[[49m[32;43m0[39;49m[43m][49m=[32m4[39m
    [38;5;66;03m# error[39;00m

[31mTypeError[39m: 'tuple' object does not support item assignment
```

Dictionaries

Mapping from keys to values:

```
Yade [25]: ende={'one':'ein' , 'two':'zwei' , 'three':'drei'}

Yade [26]: de={1:'ein' , 2:'zwei' , 3:'drei'}; en={1:'one' , 2:'two' , 3:'three'}

Yade [27]: ende['one']         ## access values
Out[27]: 'ein'

Yade [28]: de[1], en[2]
Out[28]: ('ein', 'two')
```

Functions, conditionals

```
Yade [29]: 4==5
Out[29]: False

Yade [30]: a=3.1

Yade [31]: if a<10:
   ....:     b=-2          # conditional statement
   ....: else:
   ....:     b=3
   ....:
```

(continues on next page)

(continued from previous page)

```
Yade [32]: c=0 if a<1 else 1      # trenary conditional expression

Yade [33]: b,c
Out[33]: (-2, 1)

Yade [34]: def square(x): return x**2    # define a new function
   ....:

Yade [35]: square(2)                     # and call that function
Out[35]: 4
```

Exercises

1. Read the following code and say what wil be the values of `a` and `b`:

```
a=range(5)
b=[(aa**2 if aa%2==0 else -aa**2) for aa in a]
```

Yade basics

Yade objects are constructed in the following manner (this process is also called "instantiation", since we create concrete instances of abstract classes: one individual sphere is an instance of the abstract *Sphere*, like Socrates is an instance of "man"):

```
Yade [36]: Sphere           # try also Sphere?
Out[36]: yade.wrapper.Sphere

Yade [37]: s=Sphere()       # create a Sphere, without specifying any attributes

Yade [38]: s.radius         # 'nan' is a special value meaning "not a number" (i.e.␣
 ↪not defined)
Out[38]: nan

Yade [39]: s.radius=2       # set radius of an existing object

Yade [40]: s.radius
Out[40]: 2.0

Yade [41]: ss=Sphere(radius=3)   # create Sphere, giving radius directly

Yade [42]: s.radius, ss.radius     # also try typing s.<tab> to see defined attributes
Out[42]: (2.0, 3.0)
```

Particles

Particles are the "data" component of simulation; they are the objects that will undergo some processes, though do not define those processes yet.

Singles

There is a number of pre-defined functions to create particles of certain type; in order to create a sphere, one has to (see the source of *sphere* for instance):

1. Create *Body*
2. Set *Body.shape* to be an instance of *Sphere* with some given radius
3. Set *Body.material* (last-defined material is used, otherwise a default material is created)

4. Set position and orientation in *Body.state*, compute mass and moment of inertia based on *Material* and *Shape*

In order to avoid such tasks, shorthand functions are defined in the *utils* module; to mention a few of them, they are *utils.sphere*, *utils.facet*, *utils.wall*. The *utils* module is imported at startup in such a way that the `utils.` prefix is not necessary for accessing them.

```
Yade [43]: s=sphere((0,0,0),radius=1)     # create sphere particle centered at (0,0,0)␣
↪with radius=1

Yade [44]: s.shape                        # s.shape describes the geometry of the␣
↪particle
Out[44]: <Sphere instance at 0x4ab9f00>

Yade [45]: s.shape.radius                 # we already know the Sphere class
Out[45]: 1.0

Yade [46]: s.state.mass, s.state.inertia # inertia is computed from density and␣
↪geometry
Out[46]:
(4188.790204786391,
 Vector3(1675.516081914556253,1675.516081914556253,1675.516081914556253))

Yade [47]: s.state.pos                    # position is the one we prescribed
Out[47]: Vector3(0,0,0)

Yade [48]: s2=sphere((-2,0,0),radius=1,fixed=True)      # explanation below
```

In the last example, the particle was fixed in space by the `fixed=True` parameter to *sphere*; such a particle will not move, creating a primitive boundary condition.

A particle object is not yet part of the simulation; in order to do so, a special function *O.bodies.append* (also see *Omega::bodies* and *Scene*) is called:

```
Yade [49]: O.bodies.append(s)             # adds particle s to the simulation; returns␣
↪id of the particle(s) added
Out[49]: 24
```

Packs

There are functions to generate a specific arrangement of particles in the *pack* module; for instance, cloud (random loose packing) of spheres can be generated with the *pack.SpherePack* class:

```
Yade [50]: from yade import pack

Yade [51]: sp=pack.SpherePack()                    # create an empty cloud; SpherePack␣
↪contains only geometrical information

Yade [52]: sp.makeCloud((1,1,1),(2,2,2),rMean=.2) # put spheres with defined radius␣
↪inside box given by corners (1,1,1) and (2,2,2)
Out[52]: 7

Yade [53]: for c,r in sp: print(c,r)               # print center and radius of all␣
↪particles (SpherePack is a sequence which can be iterated over)
   ....:
Vector3(1.781013445233231574,1.796053720021534739,1.286920719946763825) 0.2
Vector3(1.318145539517765474,1.688053459721743366,1.223903657135535727) 0.2
Vector3(1.711341300452232428,1.298105764717426114,1.54708711781388919) 0.2
```

(continues on next page)

(continued from previous page)

```
Vector3(1.405238241736783023,1.6361539296473111,1.682817311568887364) 0.2
Vector3(1.23776777209319655,1.26611963247597141,1.621395357195619003) 0.2
Vector3(1.317941656800318162,1.226238333661169744,1.21707159527680453) 0.2
Vector3(1.793598550161861427,1.785064960324447059,1.766165865866377471) 0.2

Yade [54]: sp.toSimulation()                          # create particles and add them to␣
↪the simulation
Out[54]: [25, 26, 27, 28, 29, 30, 31]
```

Boundaries

facet (triangle *Facet*), *wall* (infinite axes-aligned plane *Wall*) and *box* (finite axes-aligned cuboids *Box*) geometries are typically used to define boundaries. For instance, a "floor" for the simulation can be created like this:

```
Yade [55]: O.bodies.append(wall(-1,axis=2))
Out[55]: 32
```

There are other convenience functions (like *aabbWall* for creating closed or open rectangular box, or family of *ymport* functions)

Look inside

The simulation can be inspected in several ways. All data can be accessed from python directly:

```
Yade [56]: len(O.bodies)
Out[56]: 33

Yade [57]: O.bodies[10].shape.radius    # radius of body #10 (will give error if not␣
↪sphere, since only spheres have radius defined)
Out[57]: 0.16

Yade [58]: O.bodies[12].state.pos       # position of body #12
Out[58]: Vector3(1.562291348583753514,1.707455248186403152,1.031084205195870629)
```

Besides that, Yade says this at startup (the line preceding the command-line):

```
[[ ^L clears screen, ^U kills line. F12 controller, F11 3d view, F10 both, F9␣
↪generator, F8 plot. ]]
```

Controller
: Pressing `F12` brings up a window for controlling the simulation. Although typically no human intervention is done in large simulations (which run "headless", without any graphical interaction), it can be handy in small examples. There are basic information on the simulation (will be used later).

3d view
: The 3d view can be opened with F11 (or by clicking on button in the *Controller* – see below). There is a number of keyboard shortcuts to manipulate it (press `h` to get basic help), and it can be moved, rotated and zoomed using mouse. Display-related settings can be set in the "Display" tab of the controller (such as whether particles are drawn).

Inspector
: *Inspector* is opened by clicking on the appropriate button in the *Controller*. It shows (and updates) internal data of the current simulation. In particular, one can have a look at engines, particles (*Bodies*) and interactions (*Interactions*). Clicking at each of the attribute names links to the appropriate section in the documentation.

Exercises

1. What is this code going to do?

```
Yade [59]: O.bodies.append([sphere((2*i,0,0),1) for i in range(1,20)])
Out[59]: [33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
 ↪ 51]
```

2. Create a simple simulation with cloud of spheres enclosed in the box `(0,0,0)` and `(1,1,1)` with mean radius .1. (hint: *pack.SpherePack.makeCloud*)
3. Enclose the cloud created above in box with corners `(0,0,0)` and `(1,1,1)`; keep the top of the box open. (hint: *aabbWall*; type `aabbWall?` or `aabbWall??` to get help on the command line)
4. Open the 3D view, try zooming in/out; position axes so that `z` is upwards, `y` goes to the right and `x` towards you.

Engines

Engines define processes undertaken by particles. As we know from the theoretical introduction, the sequence of engines is called *simulation loop*. Let us define a simple interaction loop:

```
Yade [60]: O.engines=[                    # newlines and indentations are not␣
 ↪important until the brace is closed
   ....:     ForceResetter(),
   ....:     InsertionSortCollider([Bo1_Sphere_Aabb(),Bo1_Box_Aabb()]),
   ....:     InteractionLoop(           # dtto for the parenthesis here
   ....:         [Ig2_Sphere_Sphere_ScGeom(),Ig2_Box_Sphere_ScGeom()],
   ....:         [Ip2_FrictMat_FrictMat_FrictPhys()],
   ....:         [Law2_ScGeom_FrictPhys_CundallStrack()]
   ....:     ),
   ....:     NewtonIntegrator(damping=.2,label='newtonCustomLabel')      # define a␣
 ↪label newtonCustomLabel under which we can access this engine easily
   ....: ]
   ....:

Yade [61]: O.engines
Out[61]:
[<ForceResetter instance at 0x490bd60>,
 <InsertionSortCollider instance at 0x2f87f90>,
 <InteractionLoop instance at 0x46930e0>,
 <NewtonIntegrator instance at 0x490bba0>]

Yade [62]: O.engines[-1]==newtonCustomLabel    # is it the same object?
Out[62]: True

Yade [63]: newtonCustomLabel.damping
Out[63]: 0.2
```

Instead of typing everything into the command-line, one can describe simulation in a file (*script*) and then run yade with that file as an argument. We will therefore no longer show the command-line unless necessary; instead, only the script part will be shown. Like this:

```
O.engines=[                    # newlines and indentations are not important until the␣
 ↪brace is closed
        ForceResetter(),
        InsertionSortCollider([Bo1_Sphere_Aabb(),Bo1_Box_Aabb()]),
        InteractionLoop(           # dtto for the parenthesis here
                [Ig2_Sphere_Sphere_ScGeom(),Ig2_Box_Sphere_ScGeom()],
```

(continues on next page)

(continued from previous page)

```
                [Ip2_FrictMat_FrictMat_FrictPhys()],
                [Law2_ScGeom_FrictPhys_CundallStrack()]
        ),
        GravityEngine(gravity=(0,0,-9.81)),                    # 9.81 is the gravity␣
↪acceleration, and we say that
        NewtonIntegrator(damping=.2,label='newtonCustomLabel') # define a label under␣
↪which we can access this engine easily
]
```

Besides engines being run, it is likewise important to define how often they will run. Some engines can run only sometimes (we will see this later), while most of them will run always; the time between two successive runs of engines is *timestep* (Δt). There is a mathematical limit on the timestep value, called *critical timestep*, which is computed from properties of particles. Since there is a function for that, we can just set timestep using *PWaveTimeStep*:

```
O.dt=PWaveTimeStep()
```

Each time when the simulation loop finishes, time `O.time` is advanced by the timestep `O.dt`:

```
Yade [64]: O.dt=0.01

Yade [65]: O.time
Out[65]: 0.0

Yade [66]: O.step()

Yade [67]: O.time
Out[67]: 0.01
```

For experimenting with a single simulations, it is handy to save it to memory; this can be achieved, once everything is defined, with:

```
O.saveTmp()
```

Exercises

1. Define *engines* as in the above example, run the *Inspector* and click through the engines to see their sequence.
2. Write a simple script which will
   1. define particles as in the previous exercise (cloud of spheres inside a box open from the top) but with a smaller radius (*rMean*=.04)
   2. define a simple simulation loop, as the one given above
   3. set Δt equal to the critical P-Wave Δt
   4. save the initial simulation state to memory
3. Run the previously-defined simulation multiple times, while changing the value of timestep (use the button to reload the initial configuration).
   1. Try changing the *gravity* parameter, before running the simulation.
   2. See what happens as you increase Δt above the P-Wave value.
   3. Try changing *damping*
4. Reload the simulation, open the 3d view, open the *Inspector*, select a particle in the 3d view (shift-click). Then run the simulation and watch how forces on that particle change; pause the simulation somewhere in the middle, look at interactions of this particle.

5. At which point can we say that the deposition is done, so that the simulation can be stopped?

See also

The *Bouncing sphere* example shows a basic simulation.

1.2.3 Data mining

Read

Local data

All data of the simulation are accessible from python; when you open the *Inspector*, blue labels of various data can be clicked – left button for getting to the documentation, middle click to copy the name of the object (use `Ctrl-V` or middle-click to paste elsewhere). The interesting objects are among others (see *Omega* for a full list):

1. *O.engines*

   Engines are accessed by their index (position) in the simulation loop:

```
O.engines[0]      # first engine
O.engines[-1]     # last engine
```

   Note

   The index can change if *O.engines* is modified. *Labeling* introduced in the section below is a better solution for reliable access to a particular engine.

2. *O.bodies*

   Bodies are identified by their *id*, which is guaranteed to not change during the whole simulation:

```
O.bodies[0]                                                    # first body
[b.shape.radius for b in O.bodies if isinstance(b.shape,Sphere)]    # list of␣
↪radii of all spherical bodies
sum([b.state.mass for b in O.bodies])                          # sum of␣
↪masses of all bodies
numpy.average([b.state.vel[0] for b in O.bodies])              # average␣
↪velocity in x direction
```

   Note

   Uniqueness of *Body.id* is not guaranteed, since newly created bodies might recycle *ids* of *deleted* ones.

3. *O.forces*

   Generalized forces (forces, torques) acting on each particle. They are (usually) reset at the beginning of each step with *ForceResetter*, subsequently forces from individual interactions are accumulated in *InteractionLoop*. To access the data, use:

```
O.forces.f(0)     # force on #0
O.forces.t(1)     # torque on #1
```

4. *O.interactions*

Interactions are identified by *ids* of the respective interacting particles (they are created and deleted automatically during the simulation):

```
O.interactions[0,1]   # interactions of #0 with #1
O.interactions[1,0]   # the same object
O.bodies[0].intrs()   # all interactions of body #0
for i in O.bodies[12].intrs(): print (i.isReal,i.id1,i.id2)    # get some info␣
↪about interactions of body #12
[(i.isReal,i.id1,i.id2) for i in O.bodies[12].intrs()]         # same thing, but␣
↪make a list
```

Labels

Engines and *functors* can be *labeled*, which means that python variable of that name is automatically created.

```
Yade [1]: O.engines=[
   ...:    NewtonIntegrator(damping=.2,label='newtonCustomLabel')
   ...: ]
   ...:

Yade [2]: newtonCustomLabel.damping=.4

Yade [3]: O.engines[0].damping              # O.engines[0] and newtonCustomLabel are␣
↪the same objects
Out[3]: 0.4

Yade [4]: newtonCustomLabel==O.engines[0]   # O.engines[0] and newtonCustomLabel are␣
↪the same objects
Out[4]: True
```

Exercises

1. Find meaning of this expression:

   ```
   max([b.state.vel.norm() for b in O.bodies])
   ```

2. Run the *Gravity deposition* script, pause after a few seconds of simulation. Write expressions that compute

   1. kinetic energy $\sum \frac{1}{2} m_i |v_i|^2$
   2. average mass (hint: use numpy.average)
   3. maximum z-coordinate of all particles
   4. number of interactions of body #1

Global data

Useful measures of what happens in the simulation globally:

unbalanced force
: ratio of maximum contact force and maximum per-body force; measure of staticity, computed with *unbalancedForce*.

porosity
: ratio of void volume and total volume; computed with *porosity*.

coordination number
: average number of interactions per particle, *avgNumInteractions*

stress tensor (periodic boundary conditions)
: averaged force in interactions, computed with *normalShearStressTensors*

fabric tensor
: distribution of contacts in space (not yet implemented); can be visualized with *plotDirections*

Energies

Evaluating energy data for all components in the simulation (such as gravity work, kinetic energy, plastic dissipation, damping dissipation) can be enabled with

```
O.trackEnergy=True
```

Subsequently, energy values are accessible in the *O.energy*; it is a dictionary where its entries can be retrived with `keys()` and their values with `O.energy[key]`.

Save

PyRunner

To save data that we just learned to access, we need to call Python from within the *simulation loop*. *PyRunner* is created just for that; it inherits periodicy control from *PeriodicEngine* and takes the code to run as text (must be quoted, i.e. inside `'...'`) attribute called *command*. For instance, adding this to *O.engines* will print the current step number every one second wall clock time:

```
O.engines=O.engines+[ PyRunner(command='print(O.iter)',realPeriod=1) ]
```

Writing complicated code inside *command* is awkward; in such case, we define a function that will be called:

```
def myFunction():
        '''Print step number, and pause the simulation is unbalanced force is smaller␣
↪than 0.05.'''
        print(O.iter)
        if unbalancedForce()<0.05:
                print('Unbalanced force is smaller than 0.05, pausing.')
                O.pause()
```

Now this function can be added to *O.engines*:

```
O.engines+=[PyRunner(command='myFunction()',iterPeriod=100)]
```

or, in general, like that:

```
O.engines=[
        # ...
        PyRunner(command='myFunction()',iterPeriod=100) # call myFunction every 100␣
↪steps
]
```

Warning

If a function was declared inside a *live* yade session (ipython) and PyRunner attribute *updateGlobals is set to False* then an error `NameError: name 'myFunction' is not defined` will occur unless python globals() are updated with command

```
globals().update(locals())
```

Exercises

1. Run the *Gravity deposition* simulation, but change it such that:
    1. *utils.unbalancedForce* is printed every 2 seconds.
    2. check every 1000 steps the value of unbalanced force
        - if smaller than 0.2, set *damping* to 0.8 (hint: use labels)
        - if smaller than 0.1, pause the simulation

Keeping history

Yade provides the *plot* module used for storing and plotting variables (plotting itself will be discussed later). Let us start by importing this module and declare variable names that will be plotted:

```
from yade import plot
plot.plots={'t':('coordNum','unForce',None,'Ek')}                    # kinetic energy␣
↪will have legend on the right as indicated by None separator.
```

Periodic storing of data is done with *PyRunner* and the *plot.addData* function. Also let's enable energy tracking:

```
O.trackEnergy=True
def addPlotData():
        # this function adds current values to the history of data, under the names␣
↪specified
        plot.addData(t=O.
↪time,Ek=kineticEnergy(),coordNum=avgNumInteractions(),unForce=unbalancedForce())
```

Now this function can be added to *O.engines*:

```
O.engines+=[PyRunner(command='addPlotData()',iterPeriod=20)]
```

or, in general, like that:

```
O.engines=[  # ...,
        PyRunner(command='addPlotData()',iterPeriod=20)          # call the␣
↪addPlotData function every 20 iterations
]
```

History is stored in *plot.data*, and can be accessed using the variable name, e.g. `plot.data['Ek']`, and saved to text file (for post-processing outside yade) with *plot.saveDataTxt*.

Plot

plot provides facilities for plotting history saved with *plot.addData* as 2d plots. Data to be plotted are specified using dictionary *plot.plots*

```
plot.plots={'t':('coordNum','unForce',None,'Ek')}
```

History of all values is given as the name used for *plot.addData*; keys of the dictionary are `x`-axis values, and values are sequence of data on the `y` axis; the `None` separates data on the left and right axes (they are scaled independently). The plot itself is created with

```
plot.plot()          # on the command line, F8 can be used as shorthand
```

While the plot is open, it will be updated periodically, so that simulation evolution can be seen in real-time.

Energy plots

Plotting all energy contributions would be difficult, since names of all energies might not be known in advance. Fortunately, there is a way to handle that in Yade. It consists in two parts:

1. *plot.addData* is given all the energies that are currently defined:

   ```
   plot.addData(i=O.iter,total=O.energy.total(),**O.energy)
   ```

   The *O.energy.total* functions, which sums all energies together. The `**O.energy` is special python syntax for converting dictionary (remember that *O.energy* is a dictionary) to named functions arguments, so that the following two commands are identical:

   ```
   function(a=3,b=34)              # give arguments as arguments
   function(**{'a':3,'b':34})      # create arguments from dictionary
   ```

2. Data to plot are specified using a *function* that gives names of data to plot, rather than providing the data names directly:

   ```
   plot.plots={'i':['total']+O.energy.keys()}
   ```

   where `total` is the name we gave to `O.energy.total()` above, while `O.energy.keys()` will always return list of currently defined energies.

Energy plot example

Plotting energies inside a *live* yade session, for example by launching examples/test/triax-basic-without-plots.py would look following:

```
from yade import plot
O.trackEnergy=True
O.step()                            # performing a single simulation step is necessary␣
↪to populate O.energy.keys()
plot.plots={'t':O.energy.keys()+['total']}

def addPlotData():
        # this function adds current values to the history of data, under the names␣
↪specified
        plot.addData( t=O.time , total=O.energy.total() , **O.energy )

O.engines+=[PyRunner(command='addPlotData()',iterPeriod=20)]

globals().update(locals())          # do this only because this is an example of a live␣
↪yade session
```

Press F8 to show plot window and F11 to show 3D view, then press to start simulation.

Using multiple plots

It is also possible to make several separate plots, for example like this:

```
plot.plots={ 't':('total','kinetic') , 't ':['elastPotential','gravWork'] , 't  ':(
↪'nonviscDamp') }
```

Warning

There cannot be duplicate names declared in separate plots. This is why spaces were used above to indicate the same variable `t`.

With the caveat above, a following example inside a *live* yade session launched on examples/test/triax-basic-without-plots.py would look following:

```
from yade import plot
O.trackEnergy=True
plot.plots={ 't':('total','kinetic') , 't ':['elastPotential','gravWork'] , 't  ':(
↪'nonviscDamp') }

def addPlotData():
        # assign value to all three: 't', 't ' and 't  ' with single t=... assignment
        plot.addData( t=O.time , total=O.energy.total() , **O.energy )

O.engines+=[PyRunner(command='addPlotData()',iterPeriod=20)]

globals().update(locals())        # do this only because this is an example of a live␣
↪yade session

plot.plot(subPlots=False)         # show plots in separate windows

plot.plot(subPlots=True)          # same as pressing F8: close current plot windows␣
↪and reopen a single new one
```

Press F8 to show plot window and F11 to show 3D view, then press to start simulation, see video below:

Exercises

1. Calculate average momentum in y direction.
2. Run the *Gravity deposition* script, plotting unbalanced force and kinetic energy.
3. While the script is running, try changing the *NewtonIntegrator.damping* parameter (do it from both *Inspector* and from the command-line). What influence does it have on the evolution of unbalanced force and kinetic energy?
4. Think about and write down all energy sources (input); write down also all energy sinks (dissipation).
5. Simulate *Gravity deposition* and plot all energies as they evolve during the simulation.

See also

Most *Examples with tutorial* use plotting facilities of Yade, some of them also track energy of the simulation.

1.2.4 Setting up a simulation

See also

Examples *Gravity deposition*, *Oedometric test*, *Periodic simple shear*, *Periodic triaxial test* deal with topics discussed here.

Parametric studies

Input parameters of the simulation (such as size distribution, damping, various contact parameters, ...) influence the results, but frequently an analytical relationship is not known. To study such influence, similar simulations differing only in a few parameters can be run and results compared. Yade can be run in *batch mode*, where one simulation script is used in conjunction with *parameter table*, which specifies parameter values for each run of the script. Batch simulation are run non-interactively, i.e. without user intervention; the user must therefore start and stop the simulation explicitly.

Suppose we want to study the influence of *damping* on the evolution of kinetic energy. The script has to be adapted at several places:

1. We have to make sure the script reads relevant parameters from the *parameter table*. This is done using *utils.readParamsFromTable*; the parameters which are read are created as variables in the `yade.params.table` module:

```
readParamsFromTable(damping=.2)      # yade.params.table.damping variable will␣
↪be created
from yade.params import table              # typing table.damping is easier than␣
↪yade.params.table.damping
```

   Note that *utils.readParamsFromTable* takes default values of its parameters, which are used if the script is not run in non-batch mode.

2. Parameters from the table are used at appropriate places:

```
NewtonIntegrator(damping=table.damping),
```

3. The simulation is run non-interactively; we must therefore specify at which point it should stop:

```
O.engines+=[PyRunner(iterPeriod=1000,command='checkUnbalancedForce()')]    #␣
↪call our function defined below periodically

def checkUnbalancedForce():
   if unbalancedForce<0.05:                              # exit Yade if unbalanced␣
↪force drops below 0.05
      plot.saveDataTxt(O.tags['d.id']+'.data.bz2')      # save all data into a␣
↪unique file before exiting
      import sys
      sys.exit(0)                                       # exit the program
```

4. Finally, we must start the simulation at the very end of the script:

```
O.run()               # run forever, until stopped by checkUnbalancedForce()
waitIfBatch()         # do not finish the script until the simulation ends; does␣
↪nothing in non-batch mode
```

The *parameter table* is a simple text-file (e.g. `params.txt`), where each line specifies a simulation to run:

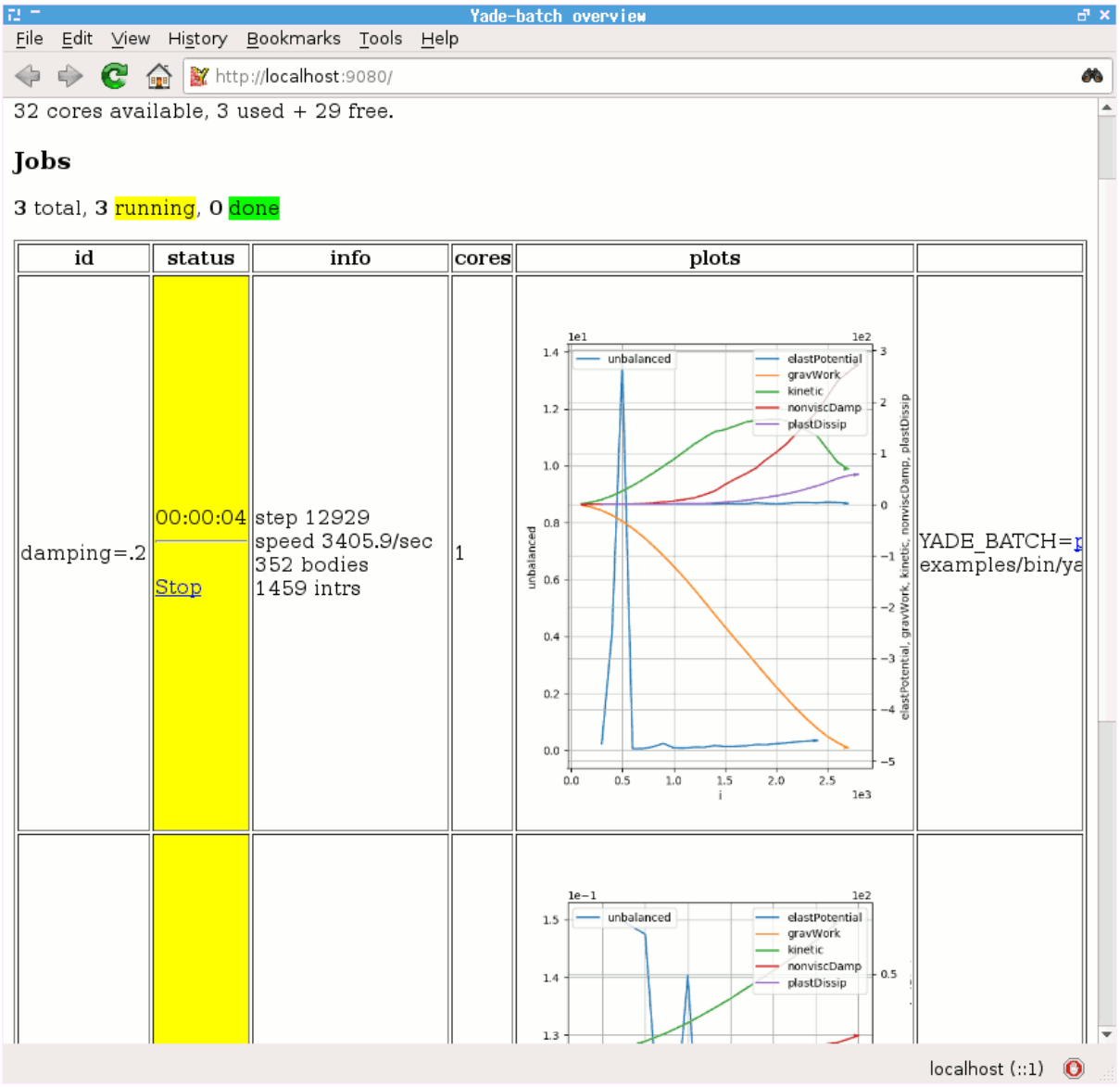
```
# comments start with # as in python
damping     # first non-comment line is variable name
.2
.4
.6
```

Finally, the simulation is run using the special batch command:

```
user@machine:~$ yade-batch params.txt simulation.py
```

Exercises

1. Run the *Gravity deposition* script in batch mode, varying *damping* to take values of `.2`, `.4`, `.6`.
2. See the http://localhost:9080 overview page while the batch is running (fig. *imgBatchExample*).



Boundary

Particles moving in infinite space usually need some constraints to make the simulation meaningful.

Supports

So far, supports (unmovable particles) were providing necessary boundary: in the *Gravity deposition* script the *geom.facetBox* is internally composed of *facets* (triangulation elements), which are `fixed` in space; facets are also used for arbitrary triangulated surfaces (see relevant sections of the *User's manual*). Another frequently used boundary is *utils.wall* (infinite axis-aligned plane).

Periodic

Periodic boundary is a "boundary" created by using periodic (rather than infinite) space. Such boundary is activated by *O.periodic=True* , and the space configuration is decribed by *O.cell* . It is well suited for studying bulk material behavior, as boundary effects are avoided, leading to smaller number of particles. On the other hand, it might not be suitable for studying localization, as any cell-level effects (such as shear bands) have to satisfy periodicity as well.

The periodic cell is described by its *reference size* of box aligned with global axes, and *current transformation*, which can capture stretch, shear and rotation. Deformation is prescribed via *velocity gradient*,

which updates the *transformation* before the next step. *Homothetic deformation* can smear *velocity gradient* accross the cell, making the boundary dissolve in the whole cell.

Stress and strains can be controlled with *PeriTriaxController*; it is possible to prescribe mixed strain/stress *goal* state using *PeriTriaxController.stressMask*.

The following creates periodic cloud of spheres and compresses to achieve σ_x=-10 kPa, σ_y=-10kPa and ε_z=-0.1. Since stress is specified for `y` and `z`, *stressMask* is binary `0b011` (x→1, y→2, z→4, in decimal 1+2=3).

```
Yade [1]: sp=pack.SpherePack()

Yade [2]: sp.makeCloud((1,1,1),(2,2,2),rMean=.16,periodic=True)
Out[2]: 20

Yade [3]: sp.toSimulation()             # implicitly sets O.periodic=True, and␣
↪O.cell.refSize to the packing period size
Out[3]: [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23]

Yade [4]: O.engines+=[PeriTriaxController(goal=(-1e4,-1e4,-.1),stressMask=0b011,
↪maxUnbalanced=.2,doneHook='functionToRunWhenFinished()')]
```

When the simulation *runs*, *PeriTriaxController* takes over the control and calls *doneHook* when *goal* is reached. A full simulation with PeriTriaxController might look like the following:

```
from yade import pack, plot

sp = pack.SpherePack()
rMean = .05
sp.makeCloud((0, 0, 0), (1, 1, 1), rMean=rMean, periodic=True)
sp.toSimulation()
O.engines = [
        ForceResetter(),
        InsertionSortCollider([Bo1_Sphere_Aabb()], verletDist=.05 * rMean),
        InteractionLoop([Ig2_Sphere_Sphere_ScGeom()],␣
↪[Ip2_FrictMat_FrictMat_FrictPhys()], [Law2_ScGeom_FrictPhys_CundallStrack()]),
        NewtonIntegrator(damping=.6),
        PeriTriaxController(
                goal=(-1e6, -1e6, -.1), stressMask=0b011, maxUnbalanced=.2, doneHook=
↪'goalReached()', label='triax', maxStrainRate=(.1, .1, .1), dynCell=True
        ),
        PyRunner(iterPeriod=100, command='addPlotData()')
]
O.dt = .5 * utils.PWaveTimeStep()
O.trackEnergy = True


def goalReached():
   print('Goal reached, strain', triax.strain, ' stress', triax.stress)
   O.pause()


def addPlotData():
   plot.addData(
           sx=triax.stress[0],
           sy=triax.stress[1],
           sz=triax.stress[2],
           ex=triax.strain[0],
           ey=triax.strain[1],
```

(continues on next page)

(continued from previous page)

```
            ez=triax.strain[2],
            i=O.iter,
            unbalanced=utils.unbalancedForce(),
            totalEnergy=O.energy.total(),
            **O.energy  # plot all energies
    )


plot.plots = {
        'i': (('unbalanced', 'go'), None, 'kinetic'),
        ' i': ('ex', 'ey', 'ez', None, 'sx', 'sy', 'sz'),
        'i ': (O.energy.keys, None, ('totalEnergy', 'bo'))
}
plot.plot()
O.saveTmp()
O.run()
```

1.2.5 Advanced & more

Particle size distribution

See *Periodic triaxial test* and examples/test/psd.py

Clumps

Clump; see *Periodic triaxial test*

Testing laws

LawTester, scripts/checks-and-tests/law-test.py

Visualization

See the example *3d-postprocessing and video recording*

- *VTKRecorder* & Paraview
- *makeVideo*
- *SnapshotEngine*
- doc/sphinx/tutorial/05-3d-postprocessing.py
- examples/test/force-network-video.py
- doc/sphinx/tutorial/make-simulation-video.py

Convert python 2 scripts to python 3

Below is a non-exhaustive list of common things to do to convert your scripts to python 3.

Mandatory:

- `print ...` becomes `print(...)`,
- `myDict.iterkeys()`, `myDict.itervalues()`, `myDict.iteritems()` becomes `myDict.keys()`, `myDict.values()`, `myDict.items()`,
- `import cPickle` becomes `import pickle`,
- `‘‘` and `<>` operators are no longer recognized,

- inconsistent use of tabs and spaces in indentation is prohibited, for this reason all scripts in yade use tabs for indentation.

Should be checked, but not always mandatory:

- (euclidian) division of two integers: `i1/i2` becomes `i1//i2`,
- `myDict.keys()`, `myDict.values()`, `myDict.items()` becomes sometimes `list(myDict.keys())`, `list(myDict.values())`, `list(myDict.items())` (depending on your usage),
- `map()`, `filter()`, `zip()` becomes sometimes `list(map())`, `list(filter())`, `list(zip())` (depending on your usage),
- string encoding is now UTF8 everywhere, it may cause problems on user inputs/outputs (keyboard, file…) with special chars.

Optional:

- `# encoding: utf-8` no longer needed

1.2.6 Examples with tutorial

The online version of this tutorial contains embedded videos.

Bouncing sphere

Following example is in file doc/sphinx/tutorial/01-bouncing-sphere.py.

```
# basic simulation showing sphere falling ball gravity,
# bouncing against another sphere representing the support

# DATA COMPONENTS

# add 2 particles to the simulation
# they the default material (utils.defaultMat)
O.bodies.append(
        [
                # fixed: particle's position in space will not change (support)
                sphere(center=(0, 0, 0), radius=.5, fixed=True),
                # this particles is free, subject to dynamics
                sphere((0, 0, 2), .5)
        ]
)

# FUNCTIONAL COMPONENTS

# simulation loop -- see presentation for the explanation
O.engines = [
        ForceResetter(),
        InsertionSortCollider([Bo1_Sphere_Aabb()]),
        InteractionLoop(
                [Ig2_Sphere_Sphere_ScGeom()],  # collision geometry
                [Ip2_FrictMat_FrictMat_FrictPhys()],  # collision "physics"
                [Law2_ScGeom_FrictPhys_CundallStrack()]  # contact law -- apply forces
        ),
        # Apply gravity force to particles. damping: numerical dissipation of energy.
        NewtonIntegrator(gravity=(0, 0, -9.81), damping=0.1)
]
```

(continues on next page)

(continued from previous page)

```
# set timestep to a fraction of the critical timestep
# the fraction is very small, so that the simulation is not too fast
# and the motion can be observed
O.dt = .5e-4 * PWaveTimeStep()

# save the simulation, so that it can be reloaded later, for experimentation
O.saveTmp()
```

Gravity deposition

Following example is in file doc/sphinx/tutorial/02-gravity-deposition.py.

```
# gravity deposition in box, showing how to plot and save history of data,
# and how to control the simulation while it is running by calling
# python functions from within the simulation loop

# import yade modules that we will use below
from yade import pack, plot

# create rectangular box from facets
O.bodies.append(geom.facetBox((.5, .5, .5), (.5, .5, .5), wallMask=31))

# create empty sphere packing
# sphere packing is not equivalent to particles in simulation, it contains only the␣
↪pure geometry
sp = pack.SpherePack()
# generate randomly spheres with uniform radius distribution
sp.makeCloud((0, 0, 0), (1, 1, 1), rMean=.05, rRelFuzz=.5)
# add the sphere pack to the simulation
sp.toSimulation()

O.engines = [
        ForceResetter(),
        InsertionSortCollider([Bo1_Sphere_Aabb(), Bo1_Facet_Aabb()]),
        InteractionLoop(
                # handle sphere+sphere and facet+sphere collisions
                [Ig2_Sphere_Sphere_ScGeom(), Ig2_Facet_Sphere_ScGeom()],
                [Ip2_FrictMat_FrictMat_FrictPhys()],
                [Law2_ScGeom_FrictPhys_CundallStrack()]
        ),
        NewtonIntegrator(gravity=(0, 0, -9.81), damping=0.4),
        # call the checkUnbalanced function (defined below) every 2 seconds
        PyRunner(command='checkUnbalanced()', realPeriod=2),
        # call the addPlotData function every 200 steps
        PyRunner(command='addPlotData()', iterPeriod=100)
]
O.dt = .5 * PWaveTimeStep()

# enable energy tracking; any simulation parts supporting it
# can create and update arbitrary energy types, which can be
# accessed as O.energy['energyName'] subsequently
O.trackEnergy = True


# if the unbalanced forces goes below .05, the packing
# is considered stabilized, therefore we stop collected
```

(continues on next page)

(continued from previous page)

```
# data history and stop
def checkUnbalanced():
   if unbalancedForce() < .05:
      O.pause()
      plot.saveDataTxt('bbb.txt.bz2')
      # plot.saveGnuplot('bbb') is also possible


# collect history of data which will be plotted
def addPlotData():
   # each item is given a names, by which it can be the unsed in plot.plots
   # the **O.energy converts dictionary-like O.energy to plot.addData arguments
   plot.addData(i=O.iter, unbalanced=unbalancedForce(), **O.energy)


# define how to plot data: 'i' (step number) on the x-axis, unbalanced force
# on the left y-axis, all energies on the right y-axis
# (O.energy.keys is function which will be called to get all defined energies)
# None separates left and right y-axis
plot.plots = {'i': ('unbalanced', None, O.energy.keys)}

# show the plot on the screen, and update while the simulation runs
plot.plot()

O.saveTmp()
```

Oedometric test

Following example is in file doc/sphinx/tutorial/03-oedometric-test.py.

```
# gravity deposition, continuing with oedometric test after stabilization
# shows also how to run parametric studies with yade-batch

# The components of the batch are:
# 1. table with parameters, one set of parameters per line (ccc.table)
# 2. readParamsFromTable which reads respective line from the parameter file
# 3. the simulation muse be run using yade-batch, not yade
#
# $ yade-batch --job-threads=1 03-oedometric-test.table 03-oedometric-test.py
#

# load parameters from file if run in batch
# default values are used if not run from batch
readParamsFromTable(rMean=.05, rRelFuzz=.3, maxLoad=1e6, minLoad=1e4)
# make rMean, rRelFuzz, maxLoad accessible directly as variables later
from yade.params.table import *

# create box with free top, and ceate loose packing inside the box
from yade import pack, plot

O.bodies.append(geom.facetBox((.5, .5, .5), (.5, .5, .5), wallMask=31))
sp = pack.SpherePack()
sp.makeCloud((0, 0, 0), (1, 1, 1), rMean=rMean, rRelFuzz=rRelFuzz)
sp.toSimulation()

O.engines = [
```

(continues on next page)

(continued from previous page)

```
        ForceResetter(),
        # sphere, facet, wall
        InsertionSortCollider([Bo1_Sphere_Aabb(), Bo1_Facet_Aabb(), Bo1_Wall_Aabb()]),
        InteractionLoop(
                # the loading plate is a wall, we need to handle sphere+sphere,␣
↪sphere+facet, sphere+wall
                [Ig2_Sphere_Sphere_ScGeom(), Ig2_Facet_Sphere_ScGeom(),␣
↪Ig2_Wall_Sphere_ScGeom()],
                [Ip2_FrictMat_FrictMat_FrictPhys()],
                [Law2_ScGeom_FrictPhys_CundallStrack()]
        ),
        NewtonIntegrator(gravity=(0, 0, -9.81), damping=0.5),
        # the label creates an automatic variable referring to this engine
        # we use it below to change its attributes from the functions called
        PyRunner(command='checkUnbalanced()', realPeriod=2, label='checker'),
]
O.dt = .5 * PWaveTimeStep()

# the following checkUnbalanced, unloadPlate and stopUnloading functions are all␣
↪called by the 'checker'
# (the last engine) one after another; this sequence defines progression of different␣
↪stages of the
# simulation, as each of the functions, when the condition is satisfied, updates
↪'checker' to call
# the next function when it is run from within the simulation next time


# check whether the gravity deposition has already finished
# if so, add wall on the top of the packing and start the oedometric test
def checkUnbalanced():
   # at the very start, unbalanced force can be low as there is only few contacts,␣
↪but it does not mean the packing is stable
   if O.iter < 5000:
      return
   # the rest will be run only if unbalanced is < .1 (stabilized packing)
   if unbalancedForce() > .1:
      return
   # add plate at the position on the top of the packing
   # the maximum finds the z-coordinate of the top of the topmost particle
   O.bodies.append(wall(max([b.state.pos[2] + b.shape.radius for b in O.bodies if␣
↪isinstance(b.shape, Sphere)]), axis=2, sense=-1))
   global plate  # without this line, the plate variable would only exist inside this␣
↪function
   plate = O.bodies[-1]  # the last particles is the plate
   # Wall objects are "fixed" by default, i.e. not subject to forces
   # prescribing a velocity will therefore make it move at constant velocity␣
↪(downwards)
   plate.state.vel = (0, 0, -.1)
   # start plotting the data now, it was not interesting before
   O.engines = O.engines + [PyRunner(command='addPlotData()', iterPeriod=200)]
   # next time, do not call this function anymore, but the next one (unloadPlate)␣
↪instead
   checker.command = 'unloadPlate()'


def unloadPlate():
```

(continues on next page)

(continued from previous page)

```
   # if the force on plate exceeds maximum load, start unloading
   if abs(O.forces.f(plate.id)[2]) > maxLoad:
      plate.state.vel *= -1
      # next time, do not call this function anymore, but the next one␣
↪(stopUnloading) instead
      checker.command = 'stopUnloading()'


def stopUnloading():
   if abs(O.forces.f(plate.id)[2]) < minLoad:
      # O.tags can be used to retrieve unique identifiers of the simulation
      # if running in batch, subsequent simulation would overwrite each other's␣
↪output files otherwise
      # d (or description) is simulation description (composed of parameter values)
      # while the id is composed of time and process number
      plot.saveDataTxt(O.tags['d.id'] + '.txt')
      O.pause()


def addPlotData():
   if not isinstance(O.bodies[-1].shape, Wall):
      plot.addData()
      return
   Fz = O.forces.f(plate.id)[2]
   plot.addData(Fz=Fz, w=plate.state.pos[2] - plate.state.refPos[2],␣
↪unbalanced=unbalancedForce(), i=O.iter)


# besides unbalanced force evolution, also plot the displacement-force diagram
plot.plots = {'i': ('unbalanced',), 'w': ('Fz',)}
plot.plot()

O.run()
# when running with yade-batch, the script must not finish until the simulation is␣
↪done fully
# this command will wait for that (has no influence in the non-batch mode)
waitIfBatch()
```

Batch table

To run the same script doc/sphinx/tutorial/03-oedometric-test.py in batch mode to test different parameters, execute command `yade-batch 03-oedometric-test.table 03-oedometric-test.py`, also visit page http://localhost:9080 to see the batch simulation progress.

```
rMean rRelFuzz maxLoad
.05 .1 1e6
.05 .2 1e6
.05 .3 1e6
```

Periodic simple shear

Following example is in file doc/sphinx/tutorial/04-periodic-simple-shear.py.

```
# encoding: utf-8

# script for periodic simple shear test, with periodic boundary
```

(continues on next page)

(continued from previous page)

```
# first compresses to attain some isotropic stress (checkStress),
# then loads in shear (checkDistorsion)
#
# the initial packing is either regular (hexagonal), with empty bands along the␣
↪boundary,
# or periodic random cloud of spheres
#
# material friction angle is initially set to zero, so that the resulting packing is␣
↪dense
# (sphere rearrangement is easier if there is no friction)
#

# setup the periodic boundary
O.periodic = True
O.cell.hSize = Matrix3(2, 0, 0, 0, 2, 0, 0, 0, 2)

from yade import pack, plot

# the "if 0:" block will be never executed, therefore the "else:" block will be
# to use cloud instead of regular packing, change to "if 1:" or something similar
if 0:
   # create cloud of spheres and insert them into the simulation
   # we give corners, mean radius, radius variation
   sp = pack.SpherePack()
   sp.makeCloud((0, 0, 0), (2, 2, 2), rMean=.1, rRelFuzz=.6, periodic=True)
   # insert the packing into the simulation
   sp.toSimulation(color=(0, 0, 1))  # pure blue
else:
   # in this case, add dense packing
   O.bodies.append(pack.regularHexa(pack.inAlignedBox((0, 0, 0), (2, 2, 2)), radius=.
↪1, gap=0, color=(0, 0, 1)))

# create "dense" packing by setting friction to zero initially
O.materials[0].frictionAngle = 0

# simulation loop (will be run at every step)
O.engines = [
        ForceResetter(),
        InsertionSortCollider([Bo1_Sphere_Aabb()]),
        InteractionLoop(
                # interaction loop
                [Ig2_Sphere_Sphere_ScGeom()],
                [Ip2_FrictMat_FrictMat_FrictPhys()],
                [Law2_ScGeom_FrictPhys_CundallStrack()]
        ),
        NewtonIntegrator(damping=.4),
        # run checkStress function (defined below) every second
        # the label is arbitrary, and is used later to refer to this engine
        PyRunner(command='checkStress()', realPeriod=1, label='checker'),
        # record data for plotting every 100 steps; addData function is defined below
        PyRunner(command='addData()', iterPeriod=100)
]

# set the integration timestep to be 1/2 of the "critical" timestep
O.dt = .5 * PWaveTimeStep()
```

(continues on next page)

(continued from previous page)

```
# prescribe isotropic normal deformation (constant strain rate)
# of the periodic cell
O.cell.velGrad = Matrix3(-.1, 0, 0, 0, -.1, 0, 0, 0, -.1)

# when to stop the isotropic compression (used inside checkStress)
limitMeanStress = -5e5


# called every second by the PyRunner engine
def checkStress():
   # stress tensor as the sum of normal and shear contributions
   # Matrix3.Zero is the intial value for sum(...)
   stress = getStress().trace() / 3.
   print('mean stress', stress)
   # if mean stress is below (bigger in absolute value) limitMeanStress, start␣
↪shearing
   if stress < limitMeanStress:
      # apply constant-rate distorsion on the periodic cell
      O.cell.velGrad = Matrix3(0, 0, .1, 0, 0, 0, 0, 0, 0)
      # change the function called by the checker engine
      # (checkStress will not be called anymore)
      checker.command = 'checkDistorsion()'
      # block rotations of particles to increase tanPhi, if desired
      # disabled by default
      if 0:
         for b in O.bodies:
            # block X,Y,Z rotations, translations are free
            b.state.blockedDOFs = 'XYZ'
            # stop rotations if any, as blockedDOFs block accelerations really
            b.state.angVel = (0, 0, 0)
      # set friction angle back to non-zero value
      # tangensOfFrictionAngle is computed by the Ip2_* functor from material
      # for future contacts change material (there is only one material for all␣
↪particles)
      O.materials[0].frictionAngle = .5  # radians
      # for existing contacts, set contact friction directly
      for i in O.interactions:
         i.phys.tangensOfFrictionAngle = tan(.5)


# called from the 'checker' engine periodically, during the shear phase
def checkDistorsion():
   # if the distorsion value is >.3, exit; otherwise do nothing
   if abs(O.cell.trsf[0, 2]) > .5:
      # save data from addData(...) before exiting into file
      # use O.tags['id'] to distinguish individual runs of the same simulation
      plot.saveDataTxt(O.tags['id'] + '.txt')
      # exit the program
      #import sys
      #sys.exit(0) # no error (0)
      O.pause()


# called periodically to store data history
def addData():
   # get the stress tensor (as 3x3 matrix)
```

(continues on next page)

(continued from previous page)

```
   stress = sum(normalShearStressTensors(), Matrix3.Zero)
   # give names to values we are interested in and save them
   plot.addData(exz=O.cell.trsf[0, 2], szz=stress[2, 2], sxz=stress[0, 2],␣
↪tanPhi=(stress[0, 2] / stress[2, 2]) if stress[2, 2] != 0 else 0, i=O.iter)
   # color particles based on rotation amount
   for b in O.bodies:
      # rot() gives rotation vector between reference and current position
      b.shape.color = scalarOnColorScale(b.state.rot().norm(), 0, pi / 2.)


# define what to plot (3 plots in total)
## exz(i), [left y axis, separate by None:] szz(i), sxz(i)
## szz(exz), sxz(exz)
## tanPhi(i)
# note the space in 'i ' so that it does not overwrite the 'i' entry
plot.plots = {'i': ('exz', None, 'szz', 'sxz'), 'exz': ('szz', 'sxz'), 'i ': ('tanPhi
↪',)}

# better show rotation of particles
Gl1_Sphere.stripes = True

# open the plot on the screen
plot.plot()

O.saveTmp()
```

3d postprocessing

Following example is in file doc/sphinx/tutorial/05-3d-postprocessing.py. This example will run for 20000 iterations, saving `*.png` snapshots, then it will make a video `3d.mpeg` out of those snapshots.

```
# demonstrate 3d postprocessing with yade
#
# 1. qt.SnapshotEngine saves images of the 3d view as it appears on the screen␣
↪periodically
#    makeVideo is then used to make real movie from those images
# 2. VTKRecorder saves data in files which can be opened with Paraview
#    see the User's manual for an intro to Paraview

# generate loose packing
from yade import pack, qt

sp = pack.SpherePack()
sp.makeCloud((0, 0, 0), (2, 2, 2), rMean=.1, rRelFuzz=.6, periodic=True)
# add to scene, make it periodic
sp.toSimulation()

O.engines = [
        ForceResetter(),
        InsertionSortCollider([Bo1_Sphere_Aabb()]),
        InteractionLoop(
                # interaction loop
                [Ig2_Sphere_Sphere_ScGeom()],
                [Ip2_FrictMat_FrictMat_FrictPhys()],
                [Law2_ScGeom_FrictPhys_CundallStrack()]
        ),
```

(continues on next page)

(continued from previous page)

```
        NewtonIntegrator(damping=.4),
        # save data for Paraview
        VTKRecorder(fileName='3d-vtk-', recorders=['all'], iterPeriod=1000),
        # save data from Yade's own 3d view
        qt.SnapshotEngine(fileBase='3d-', iterPeriod=200, label='snapshot'),
        # this engine will be called after 20000 steps, only once
        PyRunner(command='finish()', iterPeriod=20000)
]
O.dt = .5 * PWaveTimeStep()

# prescribe constant-strain deformation of the cell
O.cell.velGrad = Matrix3(-.1, 0, 0, 0, -.1, 0, 0, 0, -.1)

# we must open the view explicitly (limitation of the qt.SnapshotEngine)
qt.View()


# this function is called when the simulation is finished
def finish():
   # snapshot is label of qt.SnapshotEngine
   # the 'snapshots' attribute contains list of all saved files
   makeVideo(snapshot.snapshots, '3d.mpeg', fps=10, bps=10000)
   O.pause()


# set parameters of the renderer, to show network chains rather than particles
# these settings are accessible from the Controller window, on the second tab␣
↪("Display") as well
rr = yade.qt.Renderer()
rr.shape = False
rr.intrPhys = True
```

Periodic triaxial test

Following example is in file doc/sphinx/tutorial/06-periodic-triaxial-test.py. A variant of this exemple includes capillary forces, see doc/sphinx/tutorial/06-periodic-triaxial-test-capillarity.py

```
# encoding: utf-8

# periodic triaxial test simulation
#
# The initial packing is either
#
# 1. random cloud with uniform distribution, or
# 2. cloud with specified granulometry (radii and percentages), or
# 3. cloud of clumps, i.e. rigid aggregates of several particles
#
# The triaxial consists of 2 stages:
#
# 1. isotropic compaction, until sigmaIso is reached in all directions;
#    this stage is ended by calling compactionFinished()
# 2. constant-strain deformation along the z-axis, while maintaining
#    constant stress (sigmaIso) laterally; this stage is ended by calling
#    triaxFinished()
#
# Controlling of strain and stresses is performed via PeriTriaxController,
```

(continues on next page)

(continued from previous page)

```
# of which parameters determine type of control and also stability
# condition (maxUnbalanced) so that the packing is considered stabilized
# and the stage is done.
#

sigmaIso = -1e5

#import matplotlib
#matplotlib.use('Agg')

# generate loose packing
from yade import pack, qt, plot

O.periodic = True
sp = pack.SpherePack()
if 0:
   ## uniform distribution
   sp.makeCloud((0, 0, 0), (2, 2, 2), rMean=.1, rRelFuzz=.3, periodic=True)
else:
   ## create packing from clumps
   # configuration of one clump
   c1 = pack.SpherePack([((0, 0, 0), .03333), ((.03, 0, 0), .017), ((0, .03, 0), .
↪017)])
   # make cloud using the configuration c1 (there could c2, c3, ...; selection␣
↪between them would be random)
   sp.makeClumpCloud((0, 0, 0), (2, 2, 2), [c1], periodic=True, num=500)

# setup periodic boundary, insert the packing
sp.toSimulation()

O.engines = [
        ForceResetter(),
        InsertionSortCollider([Bo1_Sphere_Aabb()]),
        InteractionLoop([Ig2_Sphere_Sphere_ScGeom()],␣
↪[Ip2_FrictMat_FrictMat_FrictPhys()], [Law2_ScGeom_FrictPhys_CundallStrack()]),
        PeriTriaxController(
                label='triax',
                # specify target values and whether they are strains or stresses
                goal=(sigmaIso, sigmaIso, sigmaIso),
                stressMask=7,
                # type of servo-control
                dynCell=True,
                maxStrainRate=(10, 10, 10),
                # wait until the unbalanced force goes below this value
                maxUnbalanced=.1,
                relStressTol=1e-3,
                # call this function when goal is reached and the packing is stable
                doneHook='compactionFinished()'
        ),
        NewtonIntegrator(damping=.2),
        PyRunner(command='addPlotData()', iterPeriod=100),
]
O.dt = .5 * PWaveTimeStep()


def addPlotData():
```

(continues on next page)

(continued from previous page)

```
   plot.addData(
           unbalanced=unbalancedForce(),
           i=O.iter,
           sxx=triax.stress[0],
           syy=triax.stress[1],
           szz=triax.stress[2],
           exx=triax.strain[0],
           eyy=triax.strain[1],
           ezz=triax.strain[2],
           # save all available energy data
           Etot=O.energy.total(),
           **O.energy
   )


# enable energy tracking in the code
O.trackEnergy = True

# define what to plot
plot.plots = {
        'i': ('unbalanced',),
        'i ': ('sxx', 'syy', 'szz'),
        ' i': ('exx', 'eyy', 'ezz'),
        # energy plot
        ' i ': (O.energy.keys, None, 'Etot'),
}
# show the plot
plot.plot()


def compactionFinished():
   # set the current cell configuration to be the reference one
   O.cell.trsf = Matrix3.Identity
   # change control type: keep constant confinement in x,y, 20% compression in z
   triax.goal = (sigmaIso, sigmaIso, -.2)
   triax.stressMask = 3
   # allow faster deformation along x,y to better maintain stresses
   triax.maxStrainRate = (1., 1., .1)
   # next time, call triaxFinished instead of compactionFinished
   triax.doneHook = 'triaxFinished()'
   # do not wait for stabilization before calling triaxFinished
   triax.maxUnbalanced = 10


def triaxFinished():
   print('Finished')
   O.pause()
```

Fluid injection

Following example is in file doc/sphinx/tutorial/07-fluid-injection.py. The video below results from post-processing with paraview

```
# This script simulates the injection of a fluid in a localized region below immersed␣
↪particles
# The simulation is periodic along z-axis.
```

(continues on next page)

(continued from previous page)

```
# at first execution, the simulation starts by depositing the particles in the␣
↪container then saves the scene before proceeding to injection
# further execution will reload the deposited layer and start injection directly to␣
↪gain time
# WARNING: changes in some input parameters like dimensions of the box or number of␣
↪particles will not be reflected as long as the saved state is present on disk,
#          remember to erase it to force a new generation, or set newSample=True below

from yade import pack, export
import yade.timing
from math import *
from pylab import rand
import os.path
import numpy as np
import matplotlib.pyplot as plt

O.periodic = True

# Dimensions of the box and of the injection zone
width = 0.8  #
height = 1
depth = 0.4
aperture = 0.05 * width

# number of spheres
numSpheres = 2000
# contact friction during deposition
compFricDegree = 10
# porosity of the initial cloud
porosity = 0.8
# Cundall's damping (zero recommanded)
damp = 0.0
# fluid viscosity
mu = 0.01
# flow rate at the inlet
injectedFlux = -0.001
# name of output folder
key = 'output0'

newSample = False  #turn this true if you want to generate new sample by pluviation␣
↪each time you run the script

# Deduced mean size for generating the cloud and consistency check
filename = "init" + key + str(numSpheres)
volume = width * height * depth
meanRad = pow(volume * (1 - porosity) / (pi * (4 / 3.) * numSpheres), 1 / 3.)
if (meanRad * 6 > depth):
   print("INCOMPATIBLE SIZES. INCREASE DEPTH OR INCREASE NUM_SPHERES")

# if no deposited layer has been found (first execution), generate bodies
if not os.path.isfile(filename + ".yade") or newSample:  #we create new sample if it␣
↪does not exist...
   O.cell.hSize = Matrix3(width, 0, 0, 0, 3. * height, 0, 0, 0, depth)
   O.materials.append(FrictMat(young=400000.0, poisson=0.5,␣
↪frictionAngle=compFricDegree / 180.0 * pi, density=1600, label='spheres'))
   O.materials.append(FrictMat(young=400000.0, poisson=0.5,␣
```

(continues on next page)

(continued from previous page)

```
↪frictionAngle=radians(15), density=1000, label='walls'))
  lowBox = box(center=(width / 2.0, height, width / 2.0), extents=(width * 1000.0, 0,
↪ width * 1000.0), fixed=True, wire=False)
  O.bodies.append(lowBox)
  topBox = box(center=(width / 2.0, 2 * height + 4 * meanRad, width / 2.0),␣
↪extents=(width * 1000.0, 0, width * 1000.0), fixed=True, wire=False)
  O.bodies.append(topBox)

  sp = pack.SpherePack()
  sp.makeCloud((0, height + 2 * meanRad, 0), (width, 2 * height + 2 * meanRad,␣
↪depth), -1, .002, numSpheres, periodic=True, porosity=porosity, seed=2)
  O.bodies.append([sphere(s[0], s[1], color=(0.6 + 0.15 * rand(), 0.5 + 0.15 *␣
↪rand(), 0.15 + 0.15 * rand()), material='spheres') for s in sp])
else:  #... else we re-use the previous one
  O.load(filename + ".yade")

# look better
Gl1_Sphere.stripes = True

# define the fluid solver with appropriate parameter (see PeriodicFlowEngine's␣
↪documentation)
flow = PeriodicFlowEngine(
        dead=1,
        meshUpdateInterval=40,
        defTolerance=-1,
        permeabilityFactor=1.0,
        useSolver=3,
        duplicateThreshold=depth,
        wallIds=[-1, -1, 0, 1, -1, -1],
        bndCondIsPressure=[0, 0, 0, 1, 0, 0],
        viscosity=mu,
        label="flow"
)

newton = NewtonIntegrator(damping=damp, gravity=(0, -9.81, 0))

O.engines = [
        ForceResetter(),
        InsertionSortCollider([Bo1_Sphere_Aabb(), Bo1_Box_Aabb()],␣
↪allowBiggerThanPeriod=1),
        InteractionLoop([Ig2_Sphere_Sphere_ScGeom(), Ig2_Box_Sphere_ScGeom()],␣
↪[Ip2_FrictMat_FrictMat_FrictPhys()], [Law2_ScGeom_FrictPhys_CundallStrack()]),
        GlobalStiffnessTimeStepper(active=1, timeStepUpdateInterval=1000,␣
↪timestepSafetyCoefficient=0.3, defaultDt=utils.PWaveTimeStep(), label='timestepper
↪'),
        flow,  #  <====== the solver is inserted here, for the moment it is "dead"␣
↪(doing nothing)
        newton,
        # some recorders for post-processing
        PyRunner(command="flow.saveVtk(key)", iterPeriod=25, dead=1, label="vtkE"),
        VTKRecorder(recorders=["spheres", "velocity", "stress"], iterPeriod=25,␣
↪dead=1, fileName=key + '/', label="vtkR")
]

########## if this is fresh execution, get static equilibrium and save the result for␣
↪later use  ##############
```

(continues on next page)

(continued from previous page)

```
if not os.path.isfile(filename + ".yade") or newSample:
   O.run(1000, 1)
   while unbalancedForce() > 0.01:
      O.run(100, 1)
   # turn the recorders and the solver on
   vtkE.dead = vtkR.dead = flow.dead = 0
   #add a recorder and define what to plot
   O.engines = O.engines + [PyRunner(command=('myAddPlotData()'), iterPeriod=50,␣
↪label="recorder")]
   O.save(filename + ".yade")

O.saveTmp()

########## define what to plot ###############
from yade import plot
import pylab

# First, find particle at the top of the sample (for evaluating initial height of the␣
↪layer)
maxY = 0
for s in O.bodies:
   if isinstance(s.shape, Sphere):
      pos = s.state.pos
      if pos[1] > maxY:
         maxY = pos[1]


def myAddPlotData():
   index = flow.getCell(0.5 * width, height, 0.5 * depth)
   if index > 0:
      ########## find particle at the top of the sample ##############
      simpleH = 0
      for s in O.bodies:
         if isinstance(s.shape, Sphere):
            pos = s.state.pos
            if pos[1] > simpleH:
               simpleH = pos[1]
      ########## function to compute hf ##############
      cavityh = height
      for s in O.bodies:
         v = s.state.vel
         magvel = pow((v[0] * v[0] + v[1] * v[1] + v[2] * v[2]), 0.5)
         if magvel > 0.14:
            pos = s.state.pos
            if pos[1] > cavityh:
               cavityh = pos[1]

      ###########################
      plot.addData(
              t=O.time,
              i=O.iter,
              p=flow.getPorePressure((0.5 * width, height, 0.5 * depth)),
              q=-injectedFlux,
              Ho=maxY - 1,
              hf=cavityh - 1,
              H=simpleH - 1
```

(continues on next page)

(continued from previous page)

```
    )

    H = simpleH - 1
    hf = cavityh - 1


################ impose the costant flux  ###############


# In this function we find the elements of the mesh which have a face in the␣
↪injection region, to distribute the inlet flux
# it is inserted in the solver below
def imposeFlux(valF):
   found = 0
   listF1 = []
   for k in range(flow.nCells()):
      if 0 in flow.getVertices(k) and flow.getCellCenter(k)[0] > ((width - aperture) /
↪ 2.) and flow.getCellCenter(k)[0] < ((width + aperture) / 2.):
         listF1.append(k)
   flow.clearImposedFlux()
   if len(listF1) == 0:
      flow.imposeFlux((0.5 * width, height, 0.5 * depth), valF)
   else:
      for k in listF1:
         flow.imposeFlux(flow.getCellBarycenter(k), valF / float(len(listF1)))


injectedFlux = -1  #this one is large and it will fluidize violently, see below for␣
↪smoother evolutions

# very important: the flux needs to be imposed with this "hook", which plugs our␣
↪custom function in the right place in the solving sequence
flow.blockHook = "imposeFlux(injectedFlux)"

################  EXERCISE ####################
# 1- use trial and error in the shell (see the note below) to find approximately an␣
↪upper-bound of "injectedFlux", below which there is only limited movements of the␣
↪particles
# 2- implement in this script a progressive increase of the flux, starting from a␣
↪small fraction of max value above, and exceeding it by a factor 2 at the end
# 3- choose an approriate plot to show that the pressure response is initially␣
↪linear, then sublinear. Is the upper-bound from question 1 also an upper-bound for␣
↪the linear response?
# 4- compare with fig. 10 from https://doi.org/10.1103/PhysRevE.94.052905 (also␣
↪available at https://arxiv.org/pdf/1703.02319)
# 5- use paraview to generate a video similar to␣
↪https://www.youtube.com/watch?v=gH585XaQEcY (it can be with a constant flux)

#NOTE:
# to change the injected flux interactively, change it in the global scope:
# globals()["injectedFlux"]=-0.03
# else we have two variables with the same name in different scopes and
```

1.2.7 More examples

The same list with embedded videos is available online, but not recommended for viewing on slow internet connection.

A full list of examples is in file examples/list_of_examples.txt. Videos of some of those examples are listed below.

FluidCouplingLBM

- *refFastBuoyancy*, source file, video.

FluidCouplingPFV

- *refFastOedometer*, source file, video.

HydroForceEngine

- *refFastBuoyantParticles*, source file, video.
- *refFastFluidizedBed*, source file, video.
- *refFastSedimentTransportExample*, source file, video.
- *refFastLaminarShearFlow*, source file, video.
- *refFastPostProcessValidMaurin2015*, source file, video.
- *refFastValidMaurin2015*, source file, video.

PeriodicBoundaries

- *refFastCellFlipping*, source file, video.
- *refFastPeri3dController-example1*, source file, video.
- *refFastPeri3dController-shear*, source file, video.
- *refFastPeri3dController-triaxialCompression*, source file, video.
- *refFastPeriodic-compress*, source file, video.
- *refFastPeriodic-shear*, source file, video.
- *refFastPeriodic-simple-shear*, source file, video.
- *refFastPeriodic-simple*, source file, video.
- *refFastPeriodic-triax-settingHsize*, source file, video.
- *refFastPeriodic-triax*, source file, video.
- *refFastPeriodicSandPile*, source file, video.

PotentialBlocks

- *refFastWedgeYADE*, source file, video.
- *refFastCubePBscaled*, source file, video.

PotentialParticles

- *refFastCubePPscaled*, source file, video.

WireMatPM

- *refFastWirecontacttest*, source file, video.
- *refFastWirepackings*, source file, video.
- *refFastWiretensiltest*, source file, video.

Adaptiveintegrator

- *refFastSimple-scene-plot-NewtonIntegrator*, source file, video.
- *refFastSimple-scene-plot-RungeKuttaCashKarp54*, source file, video.

Agglomerate

- *refFastCompress*, source file, video.
- *refFastSimulation*, source file, video.

Baraban

- *refFastBicyclePedalEngine*, source file, video.
- *refFastBaraban*, source file, video.
- *refFastRotating-cylinder*, source file, video.

Bulldozer

- *refFastBulldozer*, source file, video.

Capillary

- *refFastCapillar*, source file, video.

CapillaryLaplaceYoung

- *refFastCapillaryPhys-example*, source file, video.
- *refFastCapillaryBridge*, source file, video.

Chained-cylinders

- *refFastCohesiveCylinderSphere*, source file, video.
- *refFastChained-cylinder-roots*, source file, video.
- *refFastChained-cylinder-spring*, source file, video.

Clumps

- *refFastAddToClump-example*, source file, video.
- *refFastApply-buoyancy-clumps*, source file, video.
- *refFastClump-hopper-test*, source file, video.
- *refFastClump-hopper-viscoelastic*, source file, video.
- *refFastClump-inbox-viscoelastic*, source file, video.
- *refFastClump-viscoelastic*, source file, video.
- *refFastReleaseFromClump-example*, source file, video.
- *refFastReplaceByClumps-example*, source file, video.
- *refFastTriax-basic-with-clumps*, source file, video.

Clumps-breakage

- *refFastClumps-breakage-first-example*, source file, video.
- *refFastAbrasive*, source file, video.
- *refFastOedometric*, source file, video.
- *refFastUniaxial-clump*, source file, video.
- *refFastUniaxial-sphere*, source file, video.

Concrete

- *refFastBrazilian*, source file, video.
- *refFastInteraction-histogram*, source file, video.
- *refFastPeriodic*, source file, video.
- *refFastTriax*, source file, video.
- *refFastUniax-post*, source file, video.
- *refFastUniax*, source file, video.

Conveyor

- *refFastConveyor*, source file, video.

Cylinders

- *refFastBendingbeams*, source file, video.
- *refFastCylinder-cylinder*, source file, video.
- *refFastCylinderconnection-roots*, source file, video.
- *refFastMikado*, source file, video.

Deformableelem

- *refFastMinimalTensileTest*, source file, video.
- *refFastTestDeformableBodies*, source file, video.
- *refFastTestDeformableBodies-pressure*, source file, video.

Grids

- *refFastCohesiveGridConnectionSphere*, source file, video.
- *refFastGridConnection-Spring*, source file, video.
- *refFastSimple-GridConnection-Falling*, source file, video.
- *refFastSimple-Grid-Falling*, source file, video.

Gts-horse

- *refFastGts-horse*, source file, video.
- *refFastGts-operators*, source file, video.
- *refFastGts-random-pack-obb*, source file, video.
- *refFastGts-random-pack*, source file, video.

Hourglass

- *refFastHourglass*, source file, video.

Packs

- *refFastPacks*, source file, video.

Pfacet

- *refFastGts-pfacet*, source file, video.
- *refFastMesh-pfacet*, source file, video.
- *refFastPFacets-grids-spheres-interacting*, source file, video.
- *refFastPfacetcreators*, source file, video.

Polyhedra

- *refFastBall*, source file, video.
- *refFastHorse*, source file, video.
- *refFastIrregular*, source file, video.
- *refFastSphere-interaction*, source file, video.
- *refFastSplitter*, source file, video.
- *refFastInteractinDetectionFactor*, source file, video.
- *refFastScGeom*, source file, video.
- *refFastTextExport*, source file, video.

PolyhedraBreak

- *refFastUniaxial-compression*, source file, video.

Ring2d

- *refFastRingCundallDamping*, source file, video.
- *refFastRingSimpleViscoelastic*, source file, video.

Rod-penetration

- *refFastModel*, source file, video.

Simple-scene

- *refFast2SpheresNormVisc*, source file, video.
- *refFastSave-then-reload*, source file, video.
- *refFastSimple-scene-default-engines*, source file, video.
- *refFastSimple-scene-energy-tracking*, source file, video.
- *refFastSimple-scene-plot*, source file, video.
- *refFastSimple-scene*, source file, video.

Stl-gts

- *refFastGts-stl*, source file, video.

Tesselationwrapper

- *refFastTesselationWrapper*, source file, video.

Test

- *refFastNet-2part-displ-unloading*, source file, video.
- *refFastNet-2part-displ*, source file, video.
- *refFastBeam-l6geom*, source file, video.
- *refFastClump-facet*, source file, video.
- *refFastClumpPack*, source file, video.
- *refFastCollider-stride-triax*, source file, video.
- *refFastCollider-stride*, source file, video.
- *refFastCombined-kinematic-engine*, source file, video.
- *refFastEnergy*, source file, video.
- *refFastFacet-box*, source file, video.
- *refFastFacet-sphere-ViscElBasic-peri*, source file, video.
- *refFastFacet-sphere-ViscElBasic*, source file, video.
- *refFastFacet-sphere*, source file, video.
- *refFastHelix*, source file, video.
- *refFastInterpolating-force*, source file, video.
- *refFastKinematic*, source file, video.
- *refFastMindlin*, source file, video.
- *refFastMulti*, source file, video.
- *refFastPack-cloud*, source file, video.
- *refFastPack-inConvexPolyhedron*, source file, video.
- *refFastPv-section*, source file, video.
- *refFastPeriodic-geom-compare*, source file, video.
- *refFastPsd*, source file, video.
- *refFastSphere-sphere-ViscElBasic-peri*, source file, video.
- *refFastSubdomain-balancer*, source file, video.
- *refFastTest-sphere-facet-corner*, source file, video.
- *refFastTest-sphere-facet*, source file, video.
- *refFastTriax-basic*, source file, video.
- *refFastTriax-basic-without-plots*, source file, video.
- *refFastUnvRead*, source file, video.

Tetra

- *refFastOneTetra*, source file, video.
- *refFastOneTetraPoly*, source file, video.
- *refFastTwoTetras*, source file, video.
- *refFastTwoTetrasPoly*, source file, video.

ViscoelasticBoundaryCondition

- *refFastViscoelasticSingleElement*, source file, video.
- *refFastViscoelasticDiscreteFoundation*, source file, video.

Chapter 2

Yade for users

2.1 DEM formulation

In this chapter, we mathematically describe general features of explicit DEM simulations, with some reference to Yade implementation of these algorithms. They are given roughly in the order as they appear in simulation; first, two particles might establish a new interaction, which consists in

1. detecting collision between particles;
2. creating new interaction and determining its properties (such as stiffness); they are either precomputed or derived from properties of both particles;

Then, for already existing interactions, the following is performed:

1. strain evaluation;
2. stress computation based on strains;
3. force application to particles in interaction.

This simplified description serves only to give meaning to the ordering of sections within this chapter. A more detailed description of this *simulation loop* is given later.

In this chapter we refer to kinematic variables of the contacts as ‘‘strains‘‘, although at this scale it is also common to speak of ‘‘displacements‘‘. Which semantic is more appropriate depends on the conceptual model one is starting from, and therefore it cannot be decided independently of specific problems. The reader familiar with displacements can mentaly replace normal strain and shear strain by normal displacement and shear displacement, respectively, without altering the meaning of what follows.

2.1.1 Collision detection

Generalities

Exact computation of collision configuration between two particles can be relatively expensive (for instance between *Sphere* and *Facet*). Taking a general pair of bodies $\mathtt{i}$ and $\mathtt{j}$ and their ‘‘exact‘‘ (In the sense of precision admissible by numerical implementation.) spatial predicates (called *Shape* in Yade) represented by point sets $P_\mathtt{i}$, $P_\mathtt{j}$ the detection generally proceeds in 2 passes:

1. fast collision detection using approximate predicate $\tilde{P}_\mathtt{i}$ and $\tilde{P}_\mathtt{j}$; they are pre-constructed in such a way as to abstract away individual features of $P_\mathtt{i}$ and $P_\mathtt{j}$ and satisfy the condition

$$\forall \mathbf{x} \in \mathbb{R}^3 : \mathbf{x} \in P_\mathtt{i} \Rightarrow \mathbf{x} \in \tilde{P}_\mathtt{i} \tag{2.1}$$

   (likewise for $P_\mathtt{j}$). The approximate predicate is called ‘‘bounding volume’’ (*Bound* in Yade) since it bounds any particle's volume from outside (by virtue of the implication). It follows that $(P_\mathtt{i} \cap P_\mathtt{j}) \neq \emptyset \Rightarrow (\tilde{P}_\mathtt{i} \cap \tilde{P}_\mathtt{j}) \neq \emptyset$ and, by applying *modus tollens*,

$$\left(\tilde{P}_\mathtt{i} \cap \tilde{P}_\mathtt{j}\right) = \emptyset \Rightarrow \left(P_\mathtt{i} \cap P_\mathtt{j}\right) = \emptyset \tag{2.2}$$

which is a candidate exclusion rule in the proper sense.

2. By filtering away impossible collisions in (2.2), a more expensive, exact collision detection algorithms can be run on possible interactions, filtering out remaining spurious couples $(\tilde{P}_i \cap \tilde{P}_j) \neq \emptyset \wedge (P_i \cap P_j) = \emptyset$. These algorithms operate on P_i and P_j and have to be able to handle all possible combinations of shape types.

It is only the first step we are concerned with here.

Algorithms

Collision evaluation algorithms have been the subject of extensive research in fields such as robotics, computer graphics and simulations. They can be roughly divided in two groups:

Hierarchical algorithms
which recursively subdivide space and restrict the number of approximate checks in the first pass, knowing that lower-level bounding volumes can intersect only if they are part of the same higher-level bounding volume. Hierarchy elements are bounding volumes of different kinds: octrees [Jung1997], bounding spheres [Hubbard1996], k-DOP's [Klosowski1998].

Flat algorithms
work directly with bounding volumes without grouping them in hierarchies first; let us only mention two kinds commonly used in particle simulations:

Sweep and prune
algorithm operates on axis-aligned bounding boxes, which overlap if and only if they overlap along all axes. These algorithms have roughly $\mathcal{O}(n \log n)$ complexity, where n is number of particles as long as they exploit *temporal coherence* of the simulation.

Grid algorithms
represent continuous R^3 space by a finite set of regularly spaced points, leading to very fast neighbor search; they can reach the $\mathcal{O}(n)$ complexity [Munjiza1998] and recent research suggests ways to overcome one of the major drawbacks of this method, which is the necessity to adjust grid cell size to the largest particle in the simulation ([Munjiza2006], the "multistep" extension).

Temporal coherence
expresses the fact that motion of particles in simulation is not arbitrary but governed by physical laws. This knowledge can be exploited to optimize performance.

Numerical stability of integrating motion equations dictates an upper limit on Δt (sect. *Stability considerations*) and, by consequence, on displacement of particles during one step. This consideration is taken into account in [Munjiza2006], implying that any particle may not move further than to a neighboring grid cell during one step allowing the $\mathcal{O}(n)$ complexity; it is also explored in the periodic variant of the sweep and prune algorithm described below.

On a finer level, it is common to enlarge $\tilde{P}_i$ predicates in such a way that they satisfy the (2.1) condition during *several* timesteps; the first collision detection pass might then be run with stride, speeding up the simulation considerably. The original publication of this optimization by Verlet [Verlet1967] used enlarged list of neighbors, giving this technique the name *Verlet list*. In general cases, however, where neighbor lists are not necessarily used, the term *Verlet distance* is employed.

Sweep and prune

Let us describe in detail the sweep and prune algorithm used for collision detection in Yade (class *InsertionSortCollider*). Axis-aligned bounding boxes (*Aabb*) are used as $\tilde{P}_i$; each *Aabb* is given by lower and upper corner $\in R^3$ (in the following, $\tilde{P}_i^{x0}$, $\tilde{P}_i^{x1}$ are minimum/maximum coordinates of $\tilde{P}_i$ along the x-axis and so on). Construction of *Aabb* from various particle *Shape*'s (such as *Sphere*, *Facet*, *Wall*) is straightforward, handled by appropriate classes deriving form *BoundFunctor* (*Bo1_Sphere_Aabb*, *Bo1_Facet_Aabb*, ...).

Presence of overlap of two *Aabb*'s can be determined from conjunction of separate overlaps of intervals

along each axis (*fig-sweep-and-prune*):

$$\left(\tilde{P}_i \cap \tilde{P}_j\right) \neq \emptyset \Leftrightarrow \bigwedge_{w\in\{x,y,z\}} \left[\left(\left(\tilde{P}_i^{w0}, \tilde{P}_i^{w1}\right) \cap \left(\tilde{P}_j^{w0}, \tilde{P}_j^{w1}\right)\right) \neq \emptyset\right]$$

where (a, b) denotes interval in R.

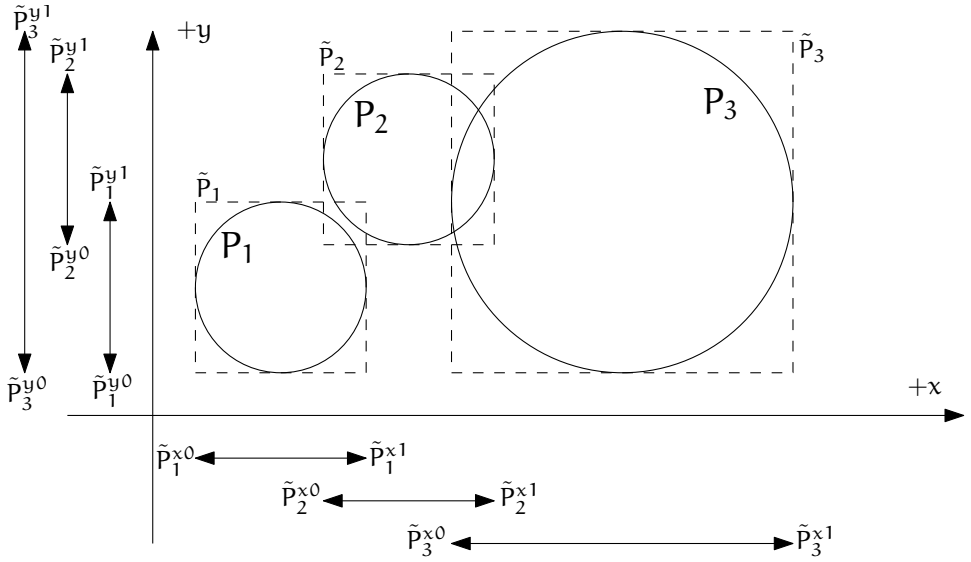


Fig. 1: Sweep and prune algorithm (shown in 2D), where *Aabb* of each sphere is represented by minimum and maximum value along each axis. Spatial overlap of *Aabb*'s is present if they overlap along all axes. In this case, $\tilde{P}_1 \cap \tilde{P}_2 \neq \emptyset$ (but note that $P_1 \cap P_2 = \emptyset$) and $\tilde{P}_2 \cap \tilde{P}_3 \neq \emptyset$.}

The collider keeps 3 separate lists (arrays) L_w for each axis $w \in \{x, y, z\}$

$$L_w = \bigcup_i \left\{\tilde{P}_i^{w0}, \tilde{P}_i^{w1}\right\}$$

where i traverses all particles. L_w arrays (sorted sets) contain respective coordinates of minimum and maximum corners for each *Aabb* (we call these coordinates *bound* in the following); besides bound, each of list elements further carries `id` referring to particle it belongs to, and a flag whether it is lower or upper bound.

In the initial step, all lists are sorted (using quicksort, average $\mathcal{O}(n \log n)$) and one axis is used to create initial interactions: the range between lower and upper bound for each body is traversed, while bounds in-between indicate potential *Aabb* overlaps which must be checked on the remaining axes as well.

At each successive step, lists are already pre-sorted. Inversions occur where a particle's coordinate has just crossed another particle's coordinate; this number is limited by numerical stability of simulation and its physical meaning (giving spatio-temporal coherence to the algorithm). The insertion sort algorithm swaps neighboring elements if they are inverted, and has complexity between $\mathcal{O}(n)$ and $\mathcal{O}(n^2)$, for pre-sorted and unsorted lists respectively. For our purposes, we need only to handle inversions, which by nature of the sort algorithm are detected inside the sort loop. An inversion might signify:

- overlap along the current axis, if an upper bound inverts (swaps) with a lower bound (i.e. that the upper bound with a higher coordinate was out of order in coming before the lower bound with a lower coordinate). Overlap along the other 2 axes is checked and if there is overlap along all axes, a new potential interaction is created.
- End of overlap along the current axis, if lower bound inverts (swaps) with an upper bound. If there is only potential interaction between the two particles in question, it is deleted.
- Nothing if both bounds are upper or both lower.

Aperiodic insertion sort

Let us show the sort algorithm on a sample sequence of numbers:

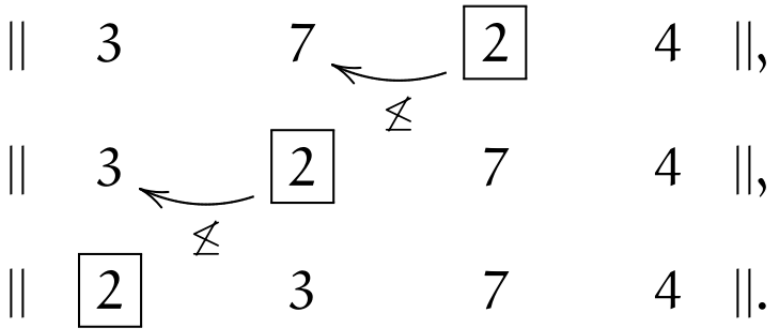
Elements are traversed from left to right; each of them keeps inverting (swapping) with neighbors to the left, moving left itself, until any of the following conditions is satisfied:

(≤)	the sorting order with the left neighbor is correct, or
(‖)	the element is at the beginning of the sequence.

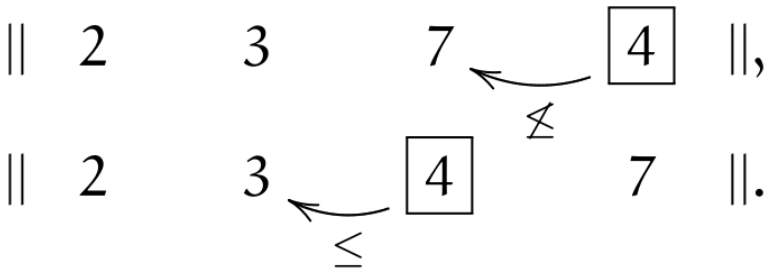
We start at the leftmost element (the current element is marked `i`)

It obviously immediately satisfies (‖), and we move to the next element:

Condition (≤) holds, therefore we move to the right. The `2` is not in order (violating (≤)) and two inversions take place; after that, (‖) holds:



The last element `4` first violates (≤), but satisfies it after one inversion



All elements having been traversed, the sequence is now sorted.

It is obvious that if the initial sequence were sorted, elements only would have to be traversed without any inversion to handle (that happens in $\mathcal{O}(n)$ time).

For each inversion during the sort in simulation, the function that investigates change in *Aabb* overlap is invoked, creating or deleting interactions.

The periodic variant of the sort algorithm is described in *Periodic insertion sort algorithm*, along with other periodic-boundary related topics.

Optimization with Verlet distances

As noted above, [Verlet1967] explored the possibility of running the collision detection only sparsely by enlarging predicates $\tilde{P}_i$.

In Yade, this is achieved by enlarging *Aabb* of particles by fixed relative length (or Verlet's distance) in all dimensions ΔL (*InsertionSortCollider.sweepLength*). Suppose the collider run last time at step m and the current step is n. *NewtonIntegrator* tracks the cummulated distance traversed by each particle between m and n by comparing the current position with the reference position from time n (*Bound::refPos*),

$$L_{mn} = |X^n - X^m| \tag{2.3}$$

triggering the collider re-run as soon as one particle gives:

$$L_{mn} > \Delta L. \tag{2.4}$$

ΔL is defined primarily by the parameter *InsertionSortCollider.verletDist*. It can be set directly by assigning a positive value, or indirectly by assigning negative value (which defines ΔL in proportion of the smallest particle radius). In addition, *InsertionSortCollider.targetInterv* can be used to adjust ΔL independently for each particle. Larger ΔL will be assigned to the fastest ones, so that all particles would ideally reach the edge of their bounds after this "target" number of iterations. Results of using Verlet distance depend highly on the nature of simulation and choice of *InsertionSortCollider.targetInterv*. Adjusting the sizes independently for each particle is especially efficient if some parts of a problem have high-speed particles will others are not moving. If it is not the case, no significant gain should be expected as compared to targetInterv=0 (assigning the same ΔL to all particles).

The number of particles and the number of available threads is also to be considered for choosing an appropriate Verlet's distance. A larger distance will result in less time spent in the collider (which runs single-threaded) and more time in computing interactions (multi-threaded). Typically, large ΔL will be used for large simulations with more than 10^5 particles on multi-core computers. On the other hand simulations with less than 10^4 particles on single processor will probably benefit from smaller ΔL. Users benchmarks may be found on Yade's wiki (see e.g. https://yade-dem.org/wiki/Colliders_performace).

2.1.2 Creating interaction between particles

Collision detection described above is only approximate. Exact collision detection depends on the geometry of individual particles and is handled separately. In Yade terminology, the *Collider* creates only *potential* interactions; potential interactions are evaluated exactly using specialized algorithms for collision of two spheres or other combinations. Exact collision detection must be run at every timestep since it is at every step that particles can change their mutual position (the collider is only run sometimes if the Verlet distance optimization is in use). Some exact collision detection algorithms are described in *Kinematic variables*; in Yade, they are implemented in classes deriving from *IGeomFunctor* (prefixed with `Ig2`).

Besides detection of geometrical overlap (which corresponds to *IGeom* in Yade), there are also non-geometrical properties of the interaction to be determined (*IPhys*). In Yade, they are computed for every new interaction by calling a functor deriving from *IPhysFunctor* (prefixed with `Ip2`) which accepts the given combination of *Material* types of both particles.

Stiffnesses

Basic DEM interaction defines two stiffnesses: normal stiffness K_N and shear (tangent) stiffness K_T. It is desirable that K_N be related to fictitious Young's modulus of the particles' material, while K_T is typically determined as a given fraction of computed K_N. The K_T/K_N ratio determines macroscopic Poisson's ratio of the arrangement, which can be shown by dimensional analysis: elastic continuum has

two parameters (E and ν) and basic DEM model also has 2 parameters with the same dimensions K_N and K_T/K_N; macroscopic Poisson's ratio is therefore determined solely by K_T/K_N and macroscopic Young's modulus is then proportional to K_N and affected by K_T/K_N.

Naturally, such analysis is highly simplifying and does not account for particle radius distribution, packing configuration and other possible parameters such as the interaction radius introduced later.

Normal stiffness

The algorithm commonly used in Yade computes normal interaction stiffness as stiffness of two springs in serial configuration with lengths equal to the sphere radii (*fig-spheres-contact-stiffness*).

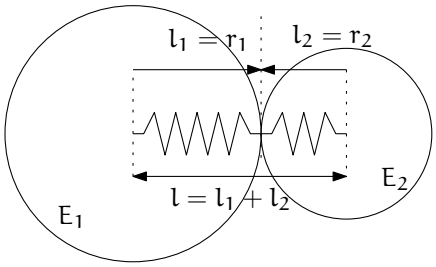


Fig. 2: Series of 2 springs representing normal stiffness of contact between 2 spheres.

Let us define distance $l = l_1 + l_2$, where l_i are distances between contact point and sphere centers, which are initially (roughly speaking) equal to sphere radii. Change of distance between the sphere centers Δl is distributed onto deformations of both spheres $\Delta l = \Delta l_1 + \Delta l_2$ proportionally to their compliances. Displacement change Δl_i generates force $F_i = K_i \Delta l_i$, where K_i assures proportionality and has physical meaning and dimension of stiffness; K_i is related to the sphere material modulus E_i and some length $\tilde{l}_i$ proportional to r_i.

$$
\begin{aligned}
\Delta l &= \Delta l_1 + \Delta l_2 \\
K_i &= E_i \tilde{l}_i \\
K_N \Delta l &= F = F_1 = F_2 \\
K_N \left(\Delta l_1 + \Delta l_2\right) &= F \\
K_N \left(\frac{F}{K_1} + \frac{F}{K_2}\right) &= F \\
K_1^{-1} + K_2^{-1} &= K_N^{-1} \\
K_N &= \frac{K_1 K_2}{K_1 + K_2} \\
K_N &= \frac{E_1 \tilde{l}_1 E_2 \tilde{l}_2}{E_1 \tilde{l}_1 + E_2 \tilde{l}_2}
\end{aligned}
$$

The most used class computing interaction properties *Ip2_FrictMat_FrictMat_FrictPhys* uses $\tilde{l}_i = 2r_i$.

Some formulations define an equivalent cross-section A_{eq}, which in that case appears in the $\tilde{l}_i$ term as $K_i = E_i \tilde{l}_i = E_i \frac{A_{eq}}{l_i}$. Such is the case for the concrete model (*Ip2_CpmMat_CpmMat_CpmPhys*), where $A_{eq} = \min(r_1, r_2)$.

For reasons given above, no pretense about equality of particle-level E_i and macroscopic modulus E should be made. Some formulations, such as [Hentz2003], introduce parameters to match them numerically. This is not appropriate, in our opinion, since it binds those values to particular features of the sphere arrangement that was used for calibration.

Other parameters

Non-elastic parameters differ for various material models. Usually, though, they are averaged from the particles' material properties, if it makes sense. For instance, *Ip2_CpmMat_CpmMat_CpmPhys* averages most quantities, while *Ip2_FrictMat_FrictMat_FrictPhys* computes internal friction angle as $\varphi = \min(\varphi_1, \varphi_2)$ to avoid friction with bodies that are frictionless.

2.1.3 Kinematic variables

In the general case, mutual configuration of two particles has 6 degrees of freedom (DoFs) just like a beam in 3D space: both particles have 6 DoFs each, but the interaction itself is free to move and rotate in space (with both spheres) having 6 DoFs itself; then $12-6=6$. They are shown at *fig-spheres-dofs*.

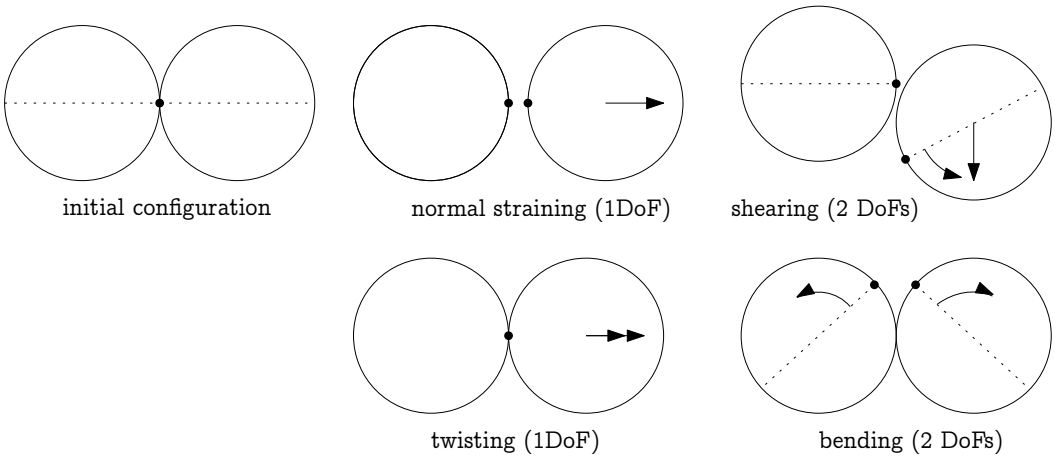


Fig. 3: Degrees of freedom of configuration of two spheres. Normal motion appears if there is a difference of linear velocity along the interaction axis ($\mathbf{n}$); shearing originates from the difference of linear velocities perpendicular to $\mathbf{n}$ *and* from the part of $\boldsymbol{\omega}_1+\boldsymbol{\omega}_2$ perpendicular to $\mathbf{n}$; twisting is caused by the part of $\boldsymbol{\omega}_1-\boldsymbol{\omega}_2$ parallel with $\mathbf{n}$; bending comes from the part of $\boldsymbol{\omega}_1-\boldsymbol{\omega}_2$ perpendicular to $\mathbf{n}$.

We will only describe normal and shear components of the relative movement in the following, leaving torsion and bending aside. The reason is that most constitutive laws for contacts do not use the latter two.

Normal deformation

Constants

Let us consider two spheres with *initial* centers $\bar{\mathbf{C}}_1$, $\bar{\mathbf{C}}_2$ and radii r_1, r_2 that enter into contact. The order of spheres within the contact is arbitrary and has no influence on the behavior. Then we define lengths

$$d_0=|\bar{\mathbf{C}}_2-\bar{\mathbf{C}}_1|$$
$$d_1=r_1+\frac{d_0-r_1-r_2}{2},\qquad d_2=d_0-d_1.$$

These quantities are *constant* throughout the life of the interaction and are computed only once when the interaction is established. The distance d_0 is the *reference distance* and is used for the conversion of absolute displacements to dimensionless strain, for instance. It is also the distance where (for usual contact laws) there is neither repulsive nor attractive force between the spheres, whence the name *equilibrium distance*.

Distances d_1 and d_2 define reduced (or expanded) radii of spheres; geometrical radii r_1 and r_2 are used only for collision detection and may not be the same as d_1 and d_2, as shown in fig. *fig-sphere-sphere*. This difference is exploited in cases where the average number of contacts between spheres should be increased, e.g. to influence the response in compression or to stabilize the packing. In such case, interactions will be created also for spheres that do not geometrically overlap based on the *interaction radius* R_I, a dimensionless parameter determining „non-locality“ of contact detection. For $R_I=1$, only spheres that touch are considered in contact; the general condition reads

$$d_0\leq R_I(r_1+r_2). \tag{2.5}$$

The value of R_I directly influences the average number of interactions per sphere (percolation), which for some models is necessary in order to achieve realistic results. In such cases, *Aabb* (or $\tilde{P}_i$ predicates in general) must be enlarged accordingly (*Bo1_Sphere_Aabb.aabbEnlargeFactor*).

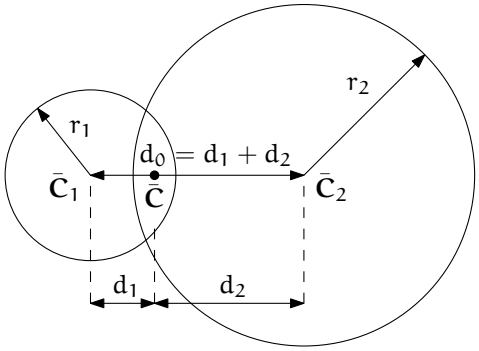


Fig. 4: Geometry of the initial contact of 2 spheres; this case pictures spheres which already overlap when the contact is created (which can be the case at the beginning of a simulation) for the sake of generality. The initial contact point $\bar{\mathbf{C}}$ is in the middle of the overlap zone.

Contact cross-section

Some constitutive laws are formulated with strains and stresses (*Law2_ScGeom_CpmPhys_Cpm*, the concrete model described later, for instance); in that case, equivalent cross-section of the contact must be introduced for the sake of dimensionality. The exact definition is rather arbitrary; the CPM model (*Ip2_CpmMat_CpmMat_CpmPhys*) uses the relation

$$A_{eq} = \pi \min(r_1, r_2)^2 \tag{2.6}$$

which will be used to convert stresses to forces, if the constitutive law used is formulated in terms of stresses and strains. Note that other values than π can be used; it will merely scale macroscopic packing stiffness; it is only for the intuitive notion of a truss-like element between the particle centers that we choose A_{eq} representing the circle area. Besides that, another function than $\min(r_1, r_2)$ can be used, although the result should depend linearly on r_1 and r_2 so that the equation gives consistent results if the particle dimensions are scaled.

Variables

The following state variables are updated as spheres undergo motion during the simulation (as $\mathbf{C}_1^\circ$ and $\mathbf{C}_2^\circ$ change):

$$\mathbf{n}^\circ = \frac{\mathbf{C}_2^\circ - \mathbf{C}_1^\circ}{|\mathbf{C}_2^\circ - \mathbf{C}_1^\circ|} \equiv \widehat{\mathbf{C}_2^\circ - \mathbf{C}_1^\circ} \tag{2.7}$$

and

$$\mathbf{C}^\circ = \mathbf{C}_1^\circ + \left(d_1 - \frac{d_0 - |\mathbf{C}_2^\circ - \mathbf{C}_1^\circ|}{2} \right) \mathbf{n}. \tag{2.8}$$

The contact point $\mathbf{C}^\circ$ is always in the middle of the spheres' overlap zone (even if the overlap is negative, when it is in the middle of the empty space between the spheres). The *contact plane* is always perpendicular to the contact plane normal $\mathbf{n}^\circ$ and passes through $\mathbf{C}^\circ$.

Normal displacement and strain can be defined as

$$u_N = |\mathbf{C}_2^\circ - \mathbf{C}_1^\circ| - d_0,$$
$$\varepsilon_N = \frac{u_N}{d_0} = \frac{|\mathbf{C}_2^\circ - \mathbf{C}_1^\circ|}{d_0} - 1.$$

Since u_N is always aligned with $\mathbf{n}$, it can be stored as a scalar value multiplied by $\mathbf{n}$ if necessary.

For massively compressive simulations, it might be beneficial to use the logarithmic strain, such that the strain tends to $-\infty$ (rather than -1) as centers of both spheres approach. Otherwise, repulsive force

would remain finite and the spheres could penetrate through each other. Therefore, we can adjust the definition of normal strain as follows:

$$\varepsilon_N = \begin{cases} \log\left(\frac{|\mathbf{C}_2^\circ - \mathbf{C}_1^\circ|}{d_0}\right) & \text{if } |\mathbf{C}_2^\circ - \mathbf{C}_1^\circ| < d_0 \\ \frac{|\mathbf{C}_2^\circ - \mathbf{C}_1^\circ|}{d_0} - 1 & \text{otherwise.} \end{cases}$$

Such definition, however, has the disadvantage of effectively increasing rigidity (up to infinity) of contacts, requiring Δt to be adjusted, lest the simulation becomes unstable. Such dynamic adjustment is possible using a stiffness-based time-stepper (*GlobalStiffnessTimeStepper* in Yade).

Shear deformation

The shear deformation (or displacement) between two particles must be computed incrementally to facilitate plasticity such as sliding in the Mohr-Coulomb friction model. Moreover, if two particles in contact are subject to a pure rigid-body motion, thereby maintaining the same relative orientation (e.g. overlap distance and shear), then $\mathbf{u}_T$ must still either track the spheres' spatial motion or must rely exclusively on sphere-local data.

Geometrical meaning of shear strain is shown in *fig-shear-2d*.

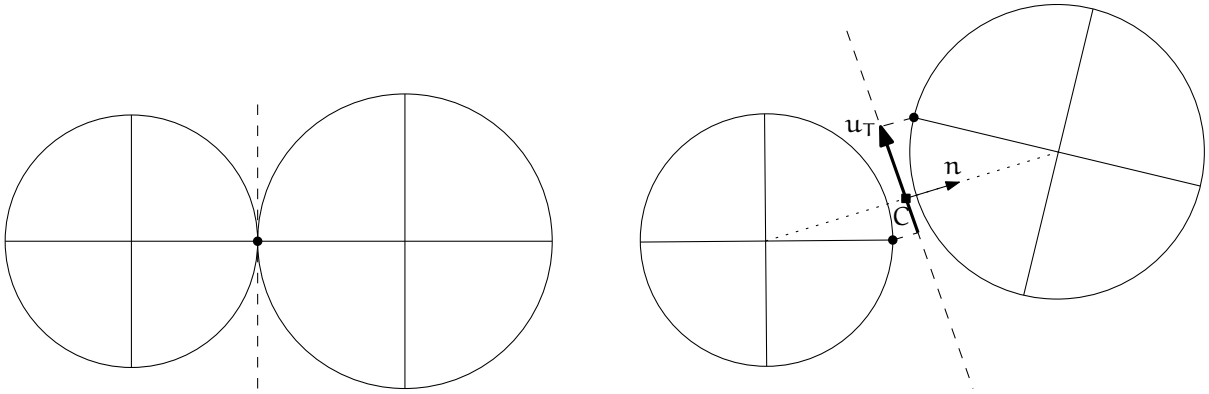


Fig. 5: Evolution of shear displacement $\mathbf{u}_T$ due to mutual motion of spheres, both linear and rotational. Left configuration is the initial contact, right configuration is after displacement and rotation of one particle.

The classical incremental algorithm is widely used in DEM codes and is described frequently ([Luding2008], [Alonso2004]). Yade implements this algorithm in the *ScGeom* class. At each step, the shear displacement $\mathbf{u}_T$ is updated; the update increment can be decomposed in 2 parts: motion of the interaction (i.e. $\mathbf{C}$ and $\mathbf{n}$) in global space and mutual motion of spheres.

1. Correcting for rigid-body motion. The orientation of the contact changes dues to changes in the spheres' positions $\mathbf{C}_1$ and $\mathbf{C}_2$, thereby updating the current $\mathbf{C}^\circ$ and $\mathbf{n}^\circ$ as per (2.7) and (2.8), as well as their orientations. The vector $\mathbf{u}_T^-$ is parallel to the contact plane at the previous step $\mathbf{n}^-$ and must be updated so that $\mathbf{u}_T^- + (\Delta\mathbf{u}_T) = \mathbf{u}_T^\circ \perp \mathbf{n}^\circ$ to correct for the tilt of the contact plane. Additionally, there can be a spin around the contact normal, which also needs to be accounted for. The update is done by projecting $\mathbf{u}_T$ onto the new contact plane first and then adding the rotation of the interaction around the new normal. Both of these are approximations, particularly the projection as it might decrease $|\mathbf{u}_T|$.

$$(\Delta\mathbf{u}_T)_1 = -\mathbf{u}_T^- \times (\mathbf{n}^- \times \mathbf{n}^\circ)$$
$$(\Delta\mathbf{u}_T)_2 = -\mathbf{u}_T^- \times \left(\frac{\Delta t}{2}\mathbf{n}^\circ \cdot (\boldsymbol{\omega}_1^\ominus + \boldsymbol{\omega}_2^\ominus)\right)\mathbf{n}^\circ$$

2. Accounting for the relative movement between the spheres, using only its part perpendicular to

$\mathbf{n}^\circ$; $\boldsymbol{v}_{12}$ denotes relative velocity of spheres at the contact point:

$$\begin{aligned}\boldsymbol{v}_{12} &= \left(\boldsymbol{v}_2^{\ominus} + \boldsymbol{\omega}_2^{\ominus} \times (-\mathrm{d}_2\mathbf{n}^\circ)\right) - \left(\boldsymbol{v}_1^{\ominus} + \boldsymbol{\omega}_1^{\ominus} \times (\mathrm{d}_1\mathbf{n}^\circ)\right)\\ \boldsymbol{v}_{12}^{\perp} &= \boldsymbol{v}_{12} - (\mathbf{n}^\circ \cdot \boldsymbol{v}_{12})\mathbf{n}^\circ\\ (\Delta\mathbf{u}_T)_3 &= -\Delta t \boldsymbol{v}_{12}^{\perp}\end{aligned}$$

Finally, we compute

$$\mathbf{u}_T^\circ = \mathbf{u}_T^- + (\Delta\mathbf{u}_T)_1 + (\Delta\mathbf{u}_T)_2 + (\Delta\mathbf{u}_T)_3.$$

2.1.4 Contact model (example)

The kinematic variables of an interaction are used to determine the forces acting on both spheres via a constitutive law. In DEM generally, some constitutive laws are expressed using strains and stresses while others prefer displacement/force formulation. The law described here falls in the latter category.

The constitutive law presented here is the most common in DEM, originally proposed by Cundall. While the kinematic variables are described in the previous section regardless of the contact model, the force evaluation depends on the nature of the material being modeled. The constitutive law presented here is the simplest non-cohesive elastic-frictional contact model, which Yade implements in *Law2_ScGeom_FrictPhys_CundallStrack* (all constitutive laws derive from base class *LawFunctor*).

When new contact is established (discussed in *Engines*) it has its properties (*IPhys*) computed from *Materials* associated with both particles. In the simple case of frictional material *FrictMat*, *Ip2_FrictMat_FrictMat_FrictPhys* creates a new *FrictPhys* instance, which defines normal stiffness K_N, shear stiffness K_T and friction angle φ.

At each step, given normal and shear displacements u_N, $\mathbf{u}_T$, normal and shear forces are computed (if $u_N > 0$, the contact is deleted without generating any forces):

$$\begin{aligned}\mathbf{F}_N &= K_N u_N \mathbf{n},\\ \mathbf{F}_T^t &= K_T \mathbf{u}_T\end{aligned}$$

where $\mathbf{F}_N$ is normal force and $\mathbf{F}_T^t$ is trial shear force. A simple non-associated stress return algorithm is applied to compute final shear force

$$\mathbf{F}_T = \begin{cases}\mathbf{F}_T^t \frac{|\mathbf{F}_N| \tan\varphi}{|\mathbf{F}_T^t|} & \text{if } |\mathbf{F}_T^t| > |\mathbf{F}_N| \tan\varphi,\\ \mathbf{F}_T^t & \text{otherwise.}\end{cases}$$

Summary force $\mathbf{F} = \mathbf{F}_N + \mathbf{F}_T$ is then applied to both particles – each particle accumulates forces and torques acting on it in the course of each step. Because the force computed acts at contact point $\mathbf{C}$, which is difference from spheres' centers, torque generated by $\mathbf{F}$ must also be considered.

$$\begin{aligned}\mathbf{F}_1 &+= \mathbf{F} & \mathbf{F}_2 &+= -\mathbf{F}\\ \mathbf{T}_1 &+= \mathrm{d}_1(-\mathbf{n}) \times \mathbf{F} & \mathbf{T}_2 &+= \mathrm{d}_2\mathbf{n} \times \mathbf{F}.\end{aligned}$$

Here, d_1 and d_2 are given by (2.1.3). Forces and torques must be applied to the same (contact) point for both particles. Otherwise, an artificial torque will appear and break conservation of angular momentum.

2.1.5 Motion integration

Each particle accumulates generalized forces (forces and torques) from the contacts in which it participates. These generalized forces are then used to integrate motion equations for each particle separately; therefore, we omit i indices denoting the i-th particle in this section.

The customary leapfrog scheme (also known as the Verlet scheme) is used, with some adjustments for rotation of non-spherical particles, as explained below. The "leapfrog" name comes from the fact that even derivatives of position/orientation are known at on-step points, whereas odd derivatives are known at mid-step points. Let us recall that we use $\mathfrak{a}^-$, $\mathfrak{a}^\circ$, $\mathfrak{a}^+$ for on-step values of $\mathfrak{a}$ at $t - \Delta t$, t and $t + \Delta t$ respectively; and $\mathfrak{a}^{\ominus}$, $\mathfrak{a}^{\oplus}$ for mid-step values of $\mathfrak{a}$ at $t - \Delta t/2$, $t + \Delta t/2$.

Described integration algorithms are implemented in the *NewtonIntegrator* class in Yade.

Position

Integrating motion consists in using current acceleration $\ddot{\mathbf{u}}^{\circ}$ on a particle to update its position from the current value $\mathbf{u}^{\circ}$ to its value at the next timestep $\mathbf{u}^{+}$. Computation of acceleration, knowing current forces $\mathbf{F}$ acting on the particle in question and its mass m, is simply

$$\ddot{\mathbf{u}}^{\circ} = \mathbf{F}/m.$$

Using the 2nd order finite difference with step Δt, we obtain

$$\ddot{\mathbf{u}}^{\circ} \cong \frac{\mathbf{u}^{-} - 2\mathbf{u}^{\circ} + \mathbf{u}^{+}}{\Delta t^2}$$

from which we express

$$\begin{aligned}\mathbf{u}^{+} &= 2\mathbf{u}^{\circ} - \mathbf{u}^{-} + \ddot{\mathbf{u}}^{\circ}\Delta t^2 = \\ &= \mathbf{u}^{\circ} + \Delta t \underbrace{\left(\frac{\mathbf{u}^{\circ} - \mathbf{u}^{-}}{\Delta t} + \ddot{\mathbf{u}}^{\circ}\Delta t\right)}_{(\dagger)}.\end{aligned}$$

Typically, $\mathbf{u}^{-}$ is already not known (only $\mathbf{u}^{\circ}$ is); we notice, however, that

$$\dot{\mathbf{u}}^{\ominus} \simeq \frac{\mathbf{u}^{\circ} - \mathbf{u}^{-}}{\Delta t},$$

i.e. the mean velocity during the previous step, which is known. Plugging this approximate into the (†) term, we also notice that mean velocity during the current step can be approximated as

$$\dot{\mathbf{u}}^{\oplus} \simeq \dot{\mathbf{u}}^{\ominus} + \ddot{\mathbf{u}}^{\circ}\Delta t,$$

which is (†); we arrive finally at

$$\mathbf{u}^{+} = \mathbf{u}^{\circ} + \Delta t\left(\dot{\mathbf{u}}^{\ominus} + \ddot{\mathbf{u}}^{\circ}\Delta t\right).$$

The algorithm can then be written down by first computing current mean velocity $\dot{\mathbf{u}}^{\oplus}$ which we need to store for the next step (just as we use its old value $\dot{\mathbf{u}}^{\ominus}$ now), then computing the position for the next time step $\mathbf{u}^{+}$:

$$\begin{aligned}\dot{\mathbf{u}}^{\oplus} &= \dot{\mathbf{u}}^{\ominus} + \ddot{\mathbf{u}}^{\circ}\Delta t \\ \mathbf{u}^{+} &= \mathbf{u}^{\circ} + \dot{\mathbf{u}}^{\oplus}\Delta t.\end{aligned}$$

Positions are known at times $i\Delta t$ (if Δt is constant) while velocities are known at $i\Delta t + \frac{\Delta t}{2}$. The fact that they interleave (jump over each other) in such way gave rise to the colloquial name "leapfrog" scheme.

Orientation

YADE has three different algorithms for integrating the rotational motion of non-spherical particles and one for spherical particles.

Orientation (spherical)

Updating particle orientation $\mathbf{q}^{\circ}$ proceeds in an analogous way to position update. First, we compute current angular acceleration $\dot{\boldsymbol{\omega}}^{\circ}$ from known current torque $\mathbf{T}$. For spherical particles where the inertia tensor is diagonal in any orientation (therefore also in current global orientation), satisfying $\mathbf{I}_{11} = \mathbf{I}_{22} = \mathbf{I}_{33}$, we can write

$$\dot{\boldsymbol{\omega}}_i^{\circ} = \mathbf{T}_i/\mathbf{I}_{11},$$

We use the same approximation scheme, obtaining an equation analogous to (2.1.5)

$$\boldsymbol{\omega}^{\oplus} = \boldsymbol{\omega}^{\ominus} + \Delta t\dot{\boldsymbol{\omega}}^{\circ}.$$

The quaternion Δq representing rotation vector $\boldsymbol{\omega}^{\oplus}\Delta t$ is constructed, i.e. such that

$$(\Delta q)_{\vartheta} = |\boldsymbol{\omega}^{\oplus}\Delta t|,$$
$$(\Delta q)_{\mathbf{u}} = \widehat{\boldsymbol{\omega}^{\oplus}\Delta t} = \widehat{\boldsymbol{\omega}^{\oplus}}$$

Finally, we compute the next orientation q^{+} by rotation composition

$$q^{+} = \Delta q q^{\circ}.$$

Orientation (aspherical)

Integrating the rotation of aspherical particles is considerably more complicated than their position, as their local reference frame is not inertial. Rotation of rigid body in the local frame, where inertia matrix $\mathbf{I}$ is diagonal, is described in the continuous form by Euler's equations ($i \in \{1,2,3\}$ and i, j, k are subsequent indices):

$$\mathbf{T}_i = \mathbf{I}_{ii}\dot{\boldsymbol{\omega}}_i + (\mathbf{I}_{kk} - \mathbf{I}_{jj})\boldsymbol{\omega}_j\boldsymbol{\omega}_k.$$

Due to the presence of both $\boldsymbol{\omega}$ and $\dot{\boldsymbol{\omega}}$, the equation cannot be solved using the standard leapfrog algorithm (that was the case for translational motion and also for the spherical bodies' rotation where this equation reduced to $\mathbf{T} = \mathbf{I}\dot{\boldsymbol{\omega}}$). The different integration algorithms for non-spherical particles can be selected using the *NewtonIntegrator.rotAlgorithm* argument of the *NewtonIntegrator*.

The default algorithm and the most accurate one was proposed by [delValle2023]. The algorithm uses a leapfrog formulation that conserves the norm of the quaternion. [Omelyan1998], a more general version of [Omelyan1999] algorithm, is also implemented. Previously, YADE used the algorithm described by [Allen1989] (pg. 84–89) and designed by [Fincham1992] for molecular dynamics problems; it consists of extending the leapfrog algorithm by mid-step/on-step estimators of quantities known at on-step/mid-step points in the basic formulation. Although it has received criticism and more precise algorithms were known (such as the previous [Omelyan1999] or also [Neto2006], [Johnson2008]), this algorithm had been implemented in Yade for its relative simplicity.

Each body has its local coordinate system based on the principal axes of inertia for that body. We use $\widetilde{\bullet}$ to denote vectors in local coordinates. The orientation of the local system is given by the current particle's orientation q° as a quaternion; this quaternion can be expressed as the (current) rotation matrix $\boldsymbol{A}$. Therefore, every vector $\mathbf{a}$ is transformed as $\widetilde{\mathbf{a}} = q\mathbf{a}q^{*} = \boldsymbol{A}\mathbf{a}$. Since $\boldsymbol{A}$ is a rotation (orthogonal) matrix, the inverse rotation $\boldsymbol{A}^{-1} = \boldsymbol{A}^{T}$.

For a given particle, we know

- $\widetilde{\mathbf{I}}^{\circ}$ (constant) inertia matrix; diagonal, since in local, principal coordinates,
- $\mathbf{T}^{\circ}$ external torque,
- q° current orientation (and its equivalent rotation matrix $\boldsymbol{A}$),
- $\boldsymbol{\omega}^{\ominus}$ mid-step angular velocity,
- $\mathbf{L}^{\ominus}$ mid-step angular momentum; this is an auxiliary variable needed in Fincham's algorithm. It will be zero in the initial step.

SPIRAL Algorithm ([delValle2023])

Our goal is to compute new values of the latter three, that is, $\mathbf{L}^{\oplus}$, q^{+}, $\boldsymbol{\omega}^{\oplus}$. We first estimate the current angular velocity:

$$\mathbf{K}_1 = dt\dot{\widetilde{\boldsymbol{\omega}}}(\widetilde{\boldsymbol{\omega}}^{\ominus}, \widetilde{\mathbf{T}}^{\circ}),$$
$$\mathbf{K}_2 = dt\dot{\widetilde{\boldsymbol{\omega}}}(\widetilde{\boldsymbol{\omega}}^{\ominus} + \mathbf{K}_1, \widetilde{\mathbf{T}}^{\circ}),$$
$$\mathbf{K}_3 = dt\dot{\widetilde{\boldsymbol{\omega}}}(\widetilde{\boldsymbol{\omega}}^{\ominus} + \frac{1}{4}(\mathbf{K}_1 + \mathbf{K}_2), \widetilde{\mathbf{T}}^{\circ}),$$
$$\widetilde{\boldsymbol{\omega}}^{\oplus} = \widetilde{\boldsymbol{\omega}}^{\ominus} + \frac{1}{6}(\mathbf{K}_1 + \mathbf{K}_2 + 4\mathbf{K}_3),$$

where $\widetilde{\boldsymbol{\omega}}$ is given by Euler's equation of motion, and we treat it as a function of angular velocity and torque. This way of integrating the angular velocity is similar to the Strong Stability Preserving Runge-Kutta-3 (SSPRK3) scheme but keeps the torque constant during the time step to avoid costly force recalculations. Then, we compute $\mathfrak{q}^+$, using $\mathfrak{q}^\circ$ and $\widetilde{\boldsymbol{\omega}}^\oplus$:

$$\mathfrak{q}^+ = \mathfrak{q}^\circ(\cos\vartheta + \frac{\widetilde{\boldsymbol{\omega}}^\oplus}{|\widetilde{\boldsymbol{\omega}}^\oplus|}\sin\vartheta),$$
$$\vartheta = \frac{dt}{2}|\widetilde{\boldsymbol{\omega}}^\oplus|,$$

where the quantity inside the parenthesis is a quaternion represented by its scalar part and its imaginary (vectorial) part. The algorithm offers a third-order approximation for both the quaternion and angular velocity calculations. As this formulation conserves the norm of the quaternion, it does not need to be normalized every time step. It is normalized every *NewtonIntegrator.normalizeEvery* steps. To finish, we compute the angular velocity and momentum in the global reference frame:

$$\boldsymbol{\omega}^\oplus = \mathbf{A}^{-1}\widetilde{\boldsymbol{\omega}}^\oplus.$$
$$\mathbf{L}^\oplus = \mathbf{A}^{-1}(\widetilde{\mathbf{I}}^\circ\widetilde{\boldsymbol{\omega}}^\oplus).$$

Omelyan Algorithm

[Omelyan1999] algorithm is also a leapfrog formulation. However, note that in a leapfrog formulation, we require the mid-step velocity and the current derivative of the velocity. But, in the case of Euler's equation, the current angular acceleration depends on the current angular velocity, which is unknown. Then, Omelyan proposes to interpolate the current angular velocity product as $\widetilde{\boldsymbol{\omega}}^\circ_j\widetilde{\boldsymbol{\omega}}^\circ_k \approx \frac{1}{2}(\widetilde{\boldsymbol{\omega}}^\ominus_j\widetilde{\boldsymbol{\omega}}^\ominus_k + \widetilde{\boldsymbol{\omega}}^\oplus_j\widetilde{\boldsymbol{\omega}}^\oplus_k)$. This leads to a non-linear system of equations that can efficiently be solved by iteration:

$$\widetilde{\boldsymbol{\omega}}^\oplus_{i,n+1} = \widetilde{\boldsymbol{\omega}}^\ominus_i + \frac{dt}{\mathbf{I}_{ii}}\left(\widetilde{\mathbf{T}}^\circ_i - \frac{1}{2}(\mathbf{I}_{kk} - \mathbf{I}_{jj})(\widetilde{\boldsymbol{\omega}}^\ominus_j\widetilde{\boldsymbol{\omega}}^\ominus_k + \widetilde{\boldsymbol{\omega}}^\oplus_{j,n}\widetilde{\boldsymbol{\omega}}^\oplus_{k,n})\right).$$

Then, we can compute the orientation of the particle with

$$\mathfrak{q}^+ = \frac{1 - \frac{dt^2}{16}|\widetilde{\boldsymbol{\omega}}^\oplus|^2}{1 + \frac{dt^2}{16}|\widetilde{\boldsymbol{\omega}}^\oplus|^2}\mathfrak{q}^\circ + \frac{dt\dot{\mathfrak{q}}^\circ}{1 + \frac{dt^2}{16}|\widetilde{\boldsymbol{\omega}}^\oplus|^2}.$$

The norm-conserving derivative of a quaternion can be calculated as

$$\dot{\mathfrak{q}}^\circ = \frac{1}{2}\mathfrak{q}^\circ\widetilde{\omega}^\oplus,$$

where $\widetilde{\omega}^\oplus$ is a quaternion with a real part equal to zero and an imaginary part equal to the angular velocity. This can also be written as

$$\begin{pmatrix}\dot{\mathfrak{q}}^\circ_w\\ \dot{\mathfrak{q}}^\circ_x\\ \dot{\mathfrak{q}}^\circ_y\\ \dot{\mathfrak{q}}^\circ_z\end{pmatrix} = \frac{1}{2}\begin{pmatrix}\mathfrak{q}^\circ_w & -\mathfrak{q}^\circ_x & -\mathfrak{q}^\circ_y & -\mathfrak{q}^\circ_z\\ \mathfrak{q}^\circ_x & \mathfrak{q}^\circ_w & -\mathfrak{q}^\circ_z & \mathfrak{q}^\circ_y\\ \mathfrak{q}^\circ_y & \mathfrak{q}^\circ_z & \mathfrak{q}^\circ_w & -\mathfrak{q}^\circ_x\\ \mathfrak{q}^\circ_z & -\mathfrak{q}^\circ_y & \mathfrak{q}^\circ_x & \mathfrak{q}^\circ_w\end{pmatrix}\begin{pmatrix}0\\ \widetilde{\boldsymbol{\omega}}^\circ_x\\ \widetilde{\boldsymbol{\omega}}^\circ_y\\ \widetilde{\boldsymbol{\omega}}^\circ_z\end{pmatrix},$$

In the same way as the last algorithm, it is a third-order approximation, and the formulation is orthonormal, meaning that the norm of the quaternion is conserved. However, this formulation is numerically not as stable as the previous one.

Fincham Algorithm

Unlike the other two algorithms, [Fincham1992] does not conserve the norm of the quaternion, which is thus automatically re-normalized at every time step, regardless of *NewtonIntegrator.normalizeEvery*. This algorithm is second-order and, unlike the two other, makes a direct use of angular momentum to deduce angular velocity in a second step. Because of this, its usage requires to assign *angular momentum*

instead of angular velocity if an initial rotational movement is to be defined for a *dynamic* body (see also *Initial (angular) velocity*). It namely goes as follows: first, we estimate the current angular momentum and compute the current local angular velocity:

$$\begin{aligned}
\mathbf{L}^{\circ} &= \mathbf{L}^{\ominus} + \mathbf{T}^{\circ}\frac{\Delta t}{2}, & \widetilde{\mathbf{L}}^{\circ} &= \mathbf{A}\mathbf{L}^{\circ},\\
\mathbf{L}^{\oplus} &= \mathbf{L}^{\ominus} + \mathbf{T}^{\circ}\Delta t, & \widetilde{\mathbf{L}}^{\oplus} &= \mathbf{A}\mathbf{L}^{\oplus},\\
\widetilde{\boldsymbol{\omega}}^{\circ} &= \widetilde{\mathbf{I}}^{\circ\,-1}\widetilde{\mathbf{L}}^{\circ}, & &\\
\widetilde{\boldsymbol{\omega}}^{\oplus} &= \widetilde{\mathbf{I}}^{\circ\,-1}\widetilde{\mathbf{L}}^{\oplus}. & &
\end{aligned}$$

Then, we evaluate $\dot{q}^{\oplus}$ from $q^{\oplus}$ and $\widetilde{\boldsymbol{\omega}}^{\oplus}$ in the same way as in (2.1.5) but shifted by $\Delta t/2$ ahead. Then we can finally compute the desired values

$$\begin{aligned}
q^{+} &= q^{\circ} + \dot{q}^{\oplus}\Delta t,\\
\boldsymbol{\omega}^{\oplus} &= \mathbf{A}^{-1}\widetilde{\boldsymbol{\omega}}^{\oplus}.
\end{aligned}$$

Clumps (rigid aggregates)

DEM simulations frequently make use of rigid aggregates of particles to model complex shapes [Price2007] called *clumps*, typically composed of many spheres. Dynamic properties of clumps are computed from the properties of its members:

- For non-overlapping clump members the clump's mass m_c is summed over members, the inertia tensor $\mathbf{I}_c$ is computed using the parallel axes theorem: $\mathbf{I}_c = \sum_i (m_i * d_i^2 + I_i)$, where m_i is the mass of clump member i, d_i is the distance from center of clump member i to clump's centroid and I_i is the inertia tensor of the clump member i.
- For overlapping clump members the clump's mass m_c is summed over cells using a regular grid spacing inside axis-aligned bounding box (*Aabb*) of the clump, the inertia tensor is computed using the parallel axes theorem: $\mathbf{I}_c = \sum_j (m_j * d_j^2 + I_j)$, where m_j is the mass of cell j, d_j is the distance from cell center to clump's centroid and I_j is the inertia tensor of the cell j.

Local axes are oriented such that they are principal and inertia tensor is diagonal and clump's orientation is changed to compensate rotation of the local system, as to not change the clump members' positions in global space. Initial positions and orientations of all clump members in local coordinate system are stored.

In Yade (class *Clump*), clump members behave as stand-alone particles during simulation for purposes of collision detection and contact resolution, except that they have no contacts created among themselves within one clump. It is at the stage of motion integration that they are treated specially. Instead of integrating each of them separately, forces/torques on those particles $\mathbf{F}_i$, $\mathbf{T}_i$ are converted to forces/torques on the clump itself. Let us denote r_i relative position of each particle with regards to clump's centroid, in global orientation. Then summary force and torque on the clump are

$$\begin{aligned}
\mathbf{F}_c &= \sum F_i,\\
\mathbf{T}_c &= \sum r_i \times F_i + T_i.
\end{aligned}$$

Motion of the clump is then integrated, using aspherical rotation integration. Afterwards, clump members are displaced in global space, to keep their initial positions and orientations in the clump's local coordinate system. In such a way, relative positions of clump members are always the same, resulting in the behavior of a rigid aggregate.

Numerical damping

In simulations of quasi-static phenomena, it it desirable to dissipate kinetic energy of particles. Since most constitutive laws (including *Law_ScGeom_FrictPhys_Basic* shown above, *Contact model (example)*) do not include velocity-based damping (such as one in [Addetta2001]), it is possible to use artificial numerical damping. The formulation is described in [Pfc3dManual30], although our version is slightly adapted. The

basic idea is to decrease forces which increase the particle velocities and vice versa by $(\Delta F)_d$, comparing the current acceleration sense and particle velocity sense. This is done by component, which makes the damping scheme clearly non-physical, as it is not invariant with respect to coordinate system rotation; on the other hand, it is very easy to compute. Cundall proposed the form (we omit particle indices i since it applies to all of them separately):

$$\frac{(\Delta F)_{dw}}{F_w} = -\lambda_d \operatorname{sgn}(F_w \dot{u}_w^{\ominus}), \quad w \in \{x, y, z\}$$

where λ_d is the damping coefficient. This formulation has several advantages [Hentz2003]:

- it acts on forces (accelerations), not constraining uniform motion;
- it is independent of eigenfrequencies of particles, they will be all damped equally;
- it needs only the dimensionless parameter λ_d which does not have to be scaled.

In Yade, we use the adapted form

$$\frac{(\Delta F)_{dw}}{F_w} = -\lambda_d \operatorname{sgn} F_w \underbrace{\left(\dot{u}_w^{\ominus} + \frac{\ddot{u}_w^{\circ} \Delta t}{2}\right)}_{\simeq \dot{u}_w^{\circ}}, \tag{2.9}$$

where we replaced the previous mid-step velocity $\dot{u}^{\ominus}$ by its on-step estimate in parentheses. This is to avoid locked-in forces that appear if the velocity changes its sign due to force application at each step, i.e. when the particle in question oscillates around the position of equilibrium with $2\Delta t$ period.

In Yade, damping (2.9) is implemented in the *NewtonIntegrator* engine; the damping coefficient λ_d is *NewtonIntegrator.damping*.

Stability considerations

Critical timestep

In order to ensure stability for the explicit integration sceheme, an upper limit is imposed on Δt:

$$\Delta t_{cr} = \frac{2}{\omega_{max}} \tag{2.10}$$

where ω_{max} is the highest eigenfrequency within the system.

Single mass-spring system

Single 1D mass-spring system with mass m and stiffness K is governed by the equation

$$m\ddot{x} = -Kx$$

where x is displacement from the mean (equilibrium) position. The solution of harmonic oscillation is $x(t) = A\cos(\omega t + \varphi)$ where phase φ and amplitude A are determined by initial conditions. The angular frequency

$$\omega^{(1)} = \sqrt{\frac{K}{m}} \tag{2.11}$$

does not depend on initial conditions. Since there is one single mass, $\omega_{max}^{(1)} = \omega^{(1)}$. Plugging (2.11) into (2.10), we obtain

$$\Delta t_{cr}^{(1)} = 2/\omega_{max}^{(1)} = 2\sqrt{m/K}$$

for a single oscillator.

General mass-spring system

In a general mass-spring system, the highest frequency occurs if two connected masses m_i, m_j are in opposite motion; let us suppose they have equal velocities (which is conservative) and they are connected by a spring with stiffness K_i: displacement Δx_i of m_i will be accompained by $\Delta x_j = -\Delta x_i$ of m_j, giving $\Delta F_i = -K_i(\Delta x_i - (-\Delta x_i)) = -2K_i\Delta x_i$. That results in apparent stiffness $K_i^{(2)} = 2K_i$, giving maximum eigenfrequency of the whole system

$$\omega_{max} = \max_i \sqrt{K_i^{(2)}/m_i}.$$

The overall critical timestep is then

$$\Delta t_{cr} = \frac{2}{\omega_{max}} = \min_i 2\sqrt{\frac{m_i}{K_i^{(2)}}} = \min_i 2\sqrt{\frac{m_i}{2K_i}} = \min_i \sqrt{2}\sqrt{\frac{m_i}{K_i}}. \tag{2.12}$$

This equation can be used for all 6 degrees of freedom (DOF) in translation and rotation, by considering generalized mass and stiffness matrices M and K, and replacing fractions $\frac{m_i}{K_i}$ by eigen values of $M.K^{-1}$. The critical timestep is then associated to the eigen mode with highest frequency :

$$\Delta t_{cr} = \min \Delta t_{crk}, \quad k \in \{1, ..., 6\}. \tag{2.13}$$

DEM simulations

In DEM simulations, per-particle stiffness $\mathbf{K}_{ij}$ is determined from the stiffnesses of contacts in which it participates. Suppose each contact has normal stiffness K_{Nk}, shear stiffness $K_{Tk} = \xi K_{Nk}$ and is oriented by normal $\mathbf{n}_k$. A translational stiffness matrix $\mathbf{K}_{ij}$ can be defined as the sum of contributions of all contacts in which it participates (indices k), as [Chareyre2005].

$$\mathbf{K}_{ij} = \sum_k (K_{Nk} - K_{Tk})\mathbf{n}_i\mathbf{n}_j + K_{Tk} = \sum_j K_{Nk}\left((1-\xi)\mathbf{n}_i\mathbf{n}_j + \xi\right) \tag{2.14}$$

with i and $j \in \{x, y, z\}$. Equations (2.13) and (2.14) determine Δt_{cr} in a simulation. A similar approach generalized to all 6 DOFs is implemented by the *GlobalStiffnessTimeStepper* engine in Yade. The derivation of generalized stiffness including rotational terms is very similar and can be found in [Aboul2017].

Note that for computation efficiency reasons, eigenvalues of the stiffness matrices are not computed. They are only approximated assuming than DOF's are uncoupled, and using the diagonal terms of $K.M^{-1}$. They give good approximates in typical mechanical systems.

There is one important condition that $\omega_{max} > 0$: if there are no contacts between particles and $\omega_{max} = 0$, we would obtain value $\Delta t_{cr} = \infty$. While formally correct, this value is numerically erroneous: we were silently supposing that stiffness remains constant during each timestep, which is not true if contacts are created as particles collide. In case of no contact, therefore, stiffness must be pre-estimated based on future interactions, as shown in the next section.

Estimation of Δt_{cr} by wave propagation speed

Estimating timestep in absence of interactions is based on the connection between interaction stiffnesses and the particle's properties. Note that in this section, symbols E and ρ refer exceptionally to Young's modulus and density of *particles*, not of macroscopic arrangement.

In Yade, particles have associated *Material* which defines density ρ (*Material.density*), and also may define (in *ElastMat* and derived classes) particle's "Young's modulus" E (*ElastMat.young*). ρ is used when particle's mass m is initially computed from its ρ, while E is taken in account when creating new interaction between particles, affecting stiffness K_N. Knowing m and K_N, we can estimate (2.14) for each particle; we obviously neglect

- number of interactions per particle N_i; for a "reasonable" radius distribution, however, there is a geometrically imposed upper limit (12 for a packing of spheres with equal radii, for instance);

- the exact relationship the between particles' rigidities E_i, E_j, supposing only that K_N is somehow proportional to them.

By defining E and ρ, particles have continuum-like quantities. Explicit integration schemes for continuum equations impose a critical timestep based on sonic speed $\sqrt{E/\rho}$; the elastic wave must not propagate farther than the minimum distance of integration points l_{min} during one step. Since E, ρ are parameters of the elastic continuum and l_{min} is fixed beforehand, we obtain

$$\Delta t_{cr}^{(c)} = l_{min}\sqrt{\frac{\rho}{E}}.$$

For our purposes, we define E and ρ for each particle separately; l_{min} can be replaced by the sphere's radius R_i; technically, $l_{min} = 2R_i$ could be used, but because of possible interactions of spheres and facets (which have zero thickness), we consider $l_{min} = R_i$ instead. Then

$$\Delta t_{cr}^{(p)} = \min_i R_i\sqrt{\frac{\rho_i}{E_i}}.$$

This algorithm is implemented in the *utils.PWaveTimeStep* function.

Let us compare this result to (2.12); this necessitates making several simplifying hypotheses:

- all particles are spherical and have the same radius R;
- the sphere's material has the same E and ρ;
- the average number of contacts per sphere is N;
- the contacts have sufficiently uniform spatial distribution around each particle;
- the $\xi = K_N/K_T$ ratio is constant for all interactions;
- contact stiffness K_N is computed from E using a formula of the form

$$K_N = E\pi' R', \tag{2.15}$$

  where π' is some constant depending on the algorithm in usefootnote{For example, $\pi' = \pi/2$ in the concrete particle model (*Ip2_CpmMat_CpmMat_CpmPhys*), while $\pi' = 2$ in the classical DEM model (*Ip2_FrictMat_FrictMat_FrictPhys*) as implemented in Yade.} and R' is half-distance between spheres in contact, equal to R for the case of interaction radius $R_I = 1$. If $R_I = 1$ (and $R' \equiv R$ by consequence), all interactions will have the same stiffness K_N. In other cases, we will consider K_N as the average stiffness computed from average R' (see below).

As all particles have the same parameters, we drop the i index in the following formulas.

We try to express the average per-particle stiffness from (2.14). It is a sum over all interactions where K_N and ξ are scalars that will not rotate with interaction, while $\mathbf{n}_w$ is w-th component of unit interaction normal $\mathbf{n}$. Since we supposed uniform spatial distribution, we can replace $\mathbf{n}_w^2$ by its average value $\overline{\mathbf{n}}_w^2$. Recognizing components of $\mathbf{n}$ as direction cosines, the average values of $\mathbf{n}_w^2$ is $1/3$. We find the average value by integrating over all possible orientations, which are uniformly distributed in space:

Moreover, since all directions are equal, we can write the per-body stiffness as $K = \mathbf{K}_w$ for all $w \in \{x, y, z\}$. We obtain

$$K = \sum K_N\left((1-\xi)\frac{1}{3} + \xi\right) = \sum K_N\frac{1+2\xi}{3}$$

and can put constant terms (everything) in front of the summation. $\sum 1$ equals the number of contacts per sphere, i.e. N. Arriving at

$$K = NK_N\frac{1-2\xi}{3},$$

we substitute K into (2.12) using (2.15):

$$\Delta t_{cr} = \sqrt{2}\sqrt{\frac{m}{K}} = \sqrt{2}\sqrt{\frac{\frac{4}{3}\pi R^3\rho}{NE\pi' R\frac{1-2\xi}{3}}} = \underbrace{R\sqrt{\frac{\rho}{E}}}_{\Delta t_{cr}^{(p)}} 2\sqrt{\frac{\pi/\pi'}{N(1-2\xi)}}.$$

The ratio of timestep $\Delta t_{cr}^{(p)}$ predicted by the p-wave velocity and numerically stable timestep Δt_{cr} is the inverse value of the last (dimensionless) term:

$$\frac{\Delta t_{cr}^{(p)}}{\Delta t_{cr}} = 2\sqrt{\frac{N(1+\xi)}{\pi/\pi'}}.$$

Actual values of this ratio depend on characteristics of packing N, $K_N/K_T = \xi$ ratio and the way of computing contact stiffness from particle rigidity. Let us show it for two models in Yade:

Concrete particle model
computes contact stiffness from the equivalent area A_{eq} first (2.6),

$$A_{eq} = \pi R^2 K_N \qquad\qquad = \frac{A_{eq} E}{d_0}.$$

d_0 is the initial contact length, which will be, for interaction radius (2.5) $R_I > 1$, in average larger than $2R$. For $R_I = 1.5$,we can roughly estimate $\overline{d}_0 = 1.25 \cdot 2R = \frac{5}{2}R$, getting

$$K_N = E\left(\frac{2}{5}\pi\right)R$$

where $\frac{2}{5}\pi = \pi'$ by comparison with (2.15).

Interaction radius $R_I = 1.5$ leads to average $N \approx 12$ interactions per sphere for dense packing of spheres with the same radius R. $\xi = 0.2$ is calibrated to match the desired macroscopic Poisson's ratio $\nu = 0.2$.

Finally, we obtain the ratio

$$\frac{\Delta t_{cr}^{(p)}}{\Delta t_{cr}} = 2\sqrt{\frac{12(1-2\cdot 0.2)}{\frac{\pi}{(2/5)\pi}}} = 3.39,$$

showing significant overestimation by the p-wave algorithm.

Non-cohesive dry friction model
is the basic model proposed by Cundall explained in *Contact model (example)*. Supposing almost-constant sphere radius R and rather dense packing, each sphere will have $N = 6$ interactions on average (that corresponds to maximally dense packing of spheres with a constant radius). If we use the *Ip2_FrictMat_FrictMat_FrictPhys* class, we have $\pi' = 2$, as $K_N = E2R$; we again use $\xi = 0.2$ (for lack of a more significant value). In this case, we obtain the result

$$\frac{\Delta t_{cr}^{(p)}}{\Delta t_{cr}} = 2\sqrt{\frac{6(1-2\cdot 0.2)}{\pi/2}} = 3.02$$

which again overestimates the numerical critical timestep.

To conclude, p-wave timestep gives estimate proportional to the real Δt_{cr}, but in the cases shown, the value of about $\Delta t = 0.3\Delta t_{cr}^{(p)}$ should be used to guarantee stable simulation.

Non-elastic Δt constraints

Let us note at this place that not only Δt_{cr} assuring numerical stability of motion integration is a constraint. In systems where particles move at relatively high velocities, position change during one timestep can lead to non-elastic irreversible effects such as damage. The Δt needed for reasonable result can be lower Δt_{cr}. We have no rigorously derived rules for such cases.

2.1.6 Periodic boundary conditions

While most DEM simulations happen in R^3 space, it is frequently useful to avoid boundary effects by using periodic space instead. In order to satisfy periodicity conditions, periodic space is created by repetition of parallelepiped-shaped cell. In Yade, periodic space is implemented in the *Cell* class. The

geometry of the cell in the reference coordinates system is defined by three edges of the parallepiped. The corresponding base vectors are stored in the columns of matrix $\mathbf{H}$ (*Cell.hSize*).

The initial $\mathbf{H}$ can be explicitly defined as a 3x3 matrix at the beginning of the simulation. There are no restricitions on the possible shapes: any parallelepiped is accepted as the initial cell. If the base vectors are axis-aligned, defining only their sizes can be more convenient than defining the full $\mathbf{H}$ matrix; in that case it is enough to define the norms of columns in $\mathbf{H}$ (see *Cell.size*).

After the definition of the initial cell's geometry, $\mathbf{H}$ should generally not be modified by direct assignment. Instead, its deformation rate will be defined via the velocity gradient *Cell.velGrad* described below. It is the only variable that let the period deformation be correctly accounted for in constitutive laws and Newton integrator (*NewtonIntegrator*).

Deformations handling

The deformation of the cell over time is defined via a tensor representing the gradient of an homogeneous velocity field $\nabla \boldsymbol{v}$ (*Cell.velGrad*). This gradient represents arbitrary combinations of rotations and stretches. It can be imposed externaly or updated by *boundary controllers* (see *PeriTriaxController* or *Peri3dController*) in order to reach target strain values or to maintain some prescribed stress.

The velocity gradient is integrated automatically over time, and the cumulated transformation is reflected in the transformation matrix $\mathbf{F}$ (*Cell.trsf*) and the current shape of the cell $\mathbf{H}$. The per-step transformation update reads (it is similar for $\mathbf{H}$), with I the identity matrix:

$$\mathbf{F}^{+} = (\mathrm{I} + \nabla \boldsymbol{v} \Delta \mathrm{t}) \mathbf{F}^{\circ}.$$

$\mathbf{F}$ is initially equal to identity and can be set back to the latter at any point in simulations, in order to define the current state as reference for strains definition in boundary controllers. It will have no effect on $\mathbf{H}$.

Along with the automatic integration of cell transformation, there is an option to homothetically displace all particles so that $\nabla \boldsymbol{v}$ is applied over the whole simulation (enabled via *Cell.homoDeform*). This avoids all boundary effects coming from change of the velocity gradient.

Collision detection in periodic cell

In usual implementations, particle positions are forced to be inside the cell by wrapping their positions if they get over the boundary (so that they appear on the other side). As we wanted to avoid abrupt changes of position (it would make particle's velocity inconsistent with step displacement change), a different method was chosen.

Approximate collision detection

Pass 1 collision detection (based on sweep and prune algorithm, sect. *Sweep and prune*) operates on axis-aligned bounding boxes (*Aabb*) of particles. During the collision detection phase, bounds of all *Aabb's* are wrapped inside the cell in the first step. At subsequent runs, every bound remembers by how many cells it was initially shifted from coordinate given by the *Aabb* and uses this offset repeatedly as it is being updated from *Aabb* during particle's motion. Bounds are sorted using the periodic insertion sort algorithm (sect. *Periodic insertion sort algorithm*), which tracks periodic cell boundary ||.

Upon inversion of two *Aabb*'s, their collision along all three axes is checked, wrapping real coordinates inside the cell for that purpose.

This algorithm detects collisions as if all particles were inside the cell but without the need of constructing "ghost particles" (to represent periodic image of a particle which enters the cell from the other side) or changing the particle's positions.

It is required by the implementation (and partly by the algorithm itself) that particles do not span more than half of the current cell size along any axis; the reason is that otherwise two (or more) contacts between both particles could appear, on each side. Since Yade identifies contacts by *Body.id* of both bodies, they would not be distinguishable.

In presence of shear, the sweep-and-prune collider could not sort bounds independently along three axes: collision along x axis depends on the mutual position of particles on the y axis. Therefore, bounding boxes *are expressed in transformed coordinates* which are perpendicular in the sense of collision detection. This requires some extra computation: *Aabb* of sphere in transformed coordinates will no longer be cube, but cuboid, as the sphere itself will appear as ellipsoid after transformation. Inversely, the sphere in simulation space will have a parallelepiped bounding "box", which is cuboid around the ellipsoid in transformed axes (the *Aabb* has axes aligned with transformed cell basis). This is shown in fig. *fig-cell-shear-aabb*.

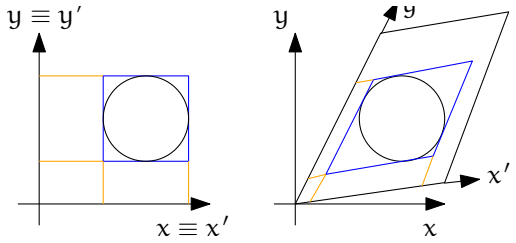


Fig. 6: Constructing axis-aligned bounding box (*Aabb*) of a sphere in simulation space coordinates (without periodic cell – left) and transformed cell coordinates (right), where collision detection axes x', y' are not identical with simulation space axes x, y. Bounds' projection to axes is shown by orange lines.

The restriction of a single particle not spanning more than half of the transformed axis becomes stringent as *Aabb* is enlarged due to shear. Considering *Aabb* of a sphere with radius r in the cell where $x' \equiv x$, $z' \equiv z$, but $\angle(y, y') = \varphi$, the x-span of the *Aabb* will be multiplied by $1/\cos\varphi$. For the infinite shear $\varphi \to \pi/2$, which can be desirable to simulate, we have $1/\cos\varphi \to \infty$. Fortunately, this limitation can be easily circumvented by realizing the quasi-identity of all periodic cells which, if repeated in space, create the same grid with their corners: the periodic cell can be flipped, keeping all particle interactions intact, as shown in fig. *fig-cell-flip*. It only necessitates adjusting the *Interaction.cellDist* of interactions and re-initialization of the collider (`Collider::invalidatePersistentData`). Cell flipping is implemented in the *Cell.flipCell* function. Automatic flip can be enabled using *Cell.flipFlippable*.

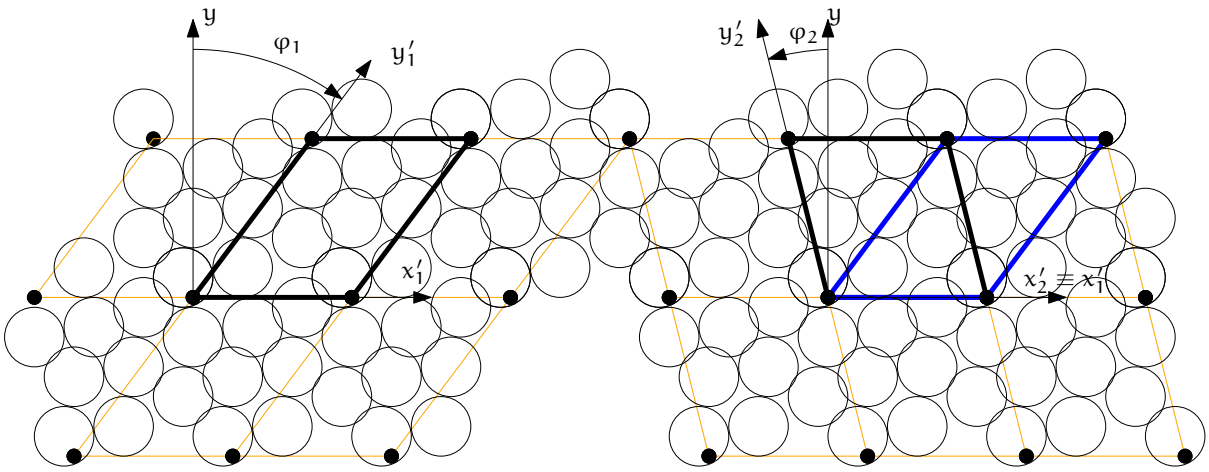


Fig. 7: Flipping cell (*Cell.flipCell*) to avoid infinite stretch of the bounding boxes' spans with growing φ. Cell flip does not affect interactions from the point of view of the simulation. The periodic arrangement on the left is the same as the one on the right, only the cell is situated differently between identical grid points of repetition; at the same time $|\varphi_2| < |\varphi_1|$ and sphere bounding box's x-span stretched by $1/\cos\varphi$ becomes smaller. Flipping can be repeated, making effective infinite shear possible.

This algorithm is implemented in *InsertionSortCollider* and is used whenever simulation is periodic (*Omega.isPeriodic*); individual *BoundFunctor's* are responsible for computing sheared *Aabb's*; currently it is implemented for spheres and facets (in *Bo1_Sphere_Aabb* and *Bo1_Facet_Aabb* respectively).

Exact collision detection

When the collider detects approximate contact (on the *Aabb* level) and the contact does not yet exist, it creates *potential* contact, which is subsequently checked by exact collision algorithms (depending on the combination of *Shapes*). Since particles can interact over many periodic cells (recall we never change their positions in simulation space), the collider embeds the relative cell coordinate of particles in the interaction itself (*Interaction.cellDist*) as an *integer* vector $\mathbf{c}$. Multiplying current cell size $\mathbf{Ts}$ by $\mathbf{c}$ component-wise, we obtain particle offset $\Delta\mathbf{x}$ in aperiodic $\mathbb{R}^3$; this value is passed (from *InteractionLoop*) to the functor computing exact collision (*IGeomFunctor*), which adds it to the position of the particle *Interaction.id2*.

By storing the integral offset $\mathbf{c}$, $\Delta\mathbf{x}$ automatically updates as cell parameters change.

Periodic insertion sort algorithm

The extension of sweep and prune algorithm (described in *Sweep and prune*) to periodic boundary conditions is non-trivial. Its cornerstone is a periodic variant of the insertion sort algorithm, which involves keeping track of the "period" of each boundary; e.g. taking period $\langle 0, 10\rangle$, then $8_1 \equiv -2_2 < 2_2$ (subscript indicating period). Doing so efficiently (without shuffling data in memory around as bound wraps from one period to another) requires moving period boundary rather than bounds themselves and making the comparison work transparently at the edge of the container.

This algorithm was also extended to handle non-orthogonal periodic *Cell* boundaries by working in transformed rather than Cartesian coordinates; this modifies computation of *Aabb* from Cartesian coordinates in which bodies are positioned (treated in detail in *Approximate collision detection*).

The sort algorithm is tracking *Aabb* extrema along all axes. At the collider's initialization, each value is assigned an integral period, i.e. its distance from the cell's interior expressed in the cell's dimension along its respective axis, and is wrapped to a value inside the cell. We put the period number in subscript.

Let us give an example of coordinate sequence along x axis (in a real case, the number of elements would be even, as there is maximum and minimum value couple for each particle; this demonstration only shows the sorting algorithm, however.)

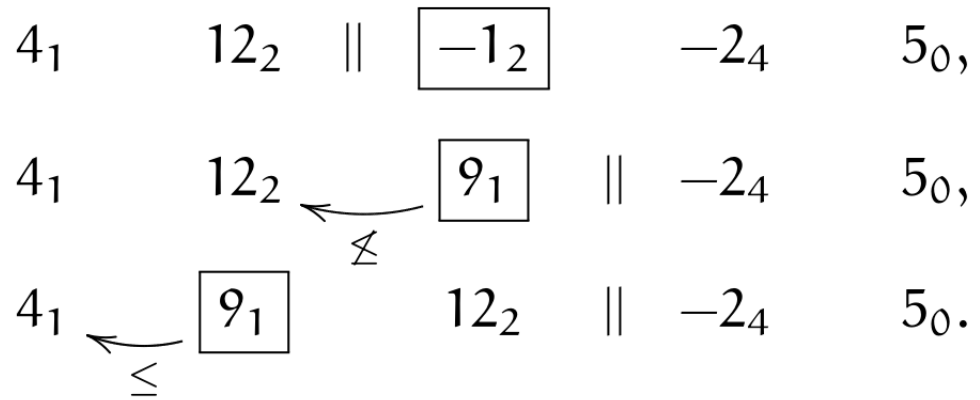
$$4_1 \qquad 12_2 \quad \| \quad -1_2 \qquad -2_4 \qquad 5_0$$

with cell x-size $s_x = 10$. The 4_1 value then means that the real coordinate x_i of this extremum is $x_i + 1 \cdot 10 = 4$, i.e. $x_i = -4$. The $\|$ symbol denotes the periodic cell boundary.

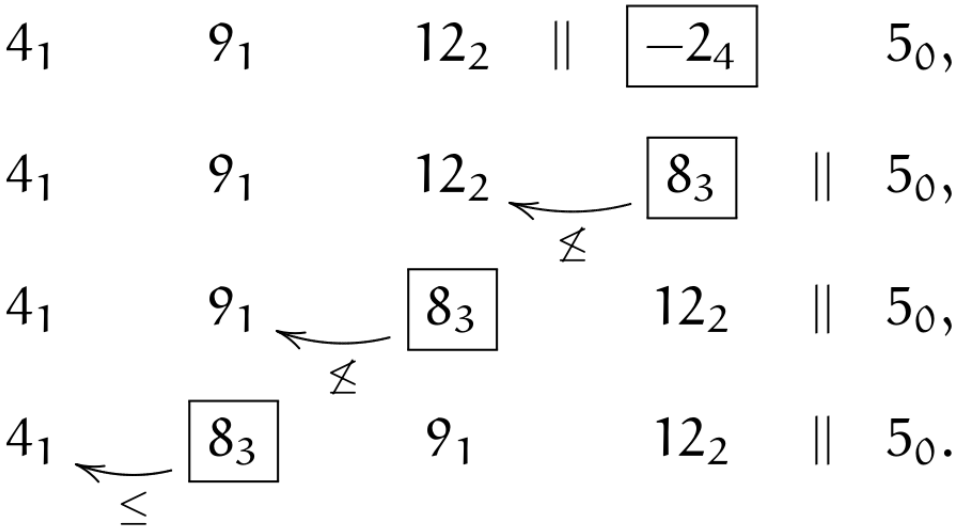
Sorting starts from the first element in the cell, i.e. right of $\|$, and inverts elements as in the aperiodic variant. The rules are, however, more complicated due to the presence of the boundary $\|$:

$(\leq)$	stop inverting if neighbors are ordered;
$(\|\bullet)$	current element left of $\|$ is below 0 (lower period boundary); in this case, decrement element's period, decrease its coordinate by s_x and move $\|$ right;
$(\bullet\|)$	current element right of $\|$ is above s_x (upper period boundary); increment element's period, increase its coordinate by s_x and move $\|$ left;
$(\nleftrightarrow)$	inversion across $\|$ must subtract s_x from the left coordinate during comparison. If the elements are not in order, they are swapped, but they must have their periods changed as they traverse $\|$. Apply $(\|\circ)$ if necessary;
$(\|\circ)$	if after $(\nleftrightarrow)$ the element that is now right of $\|$ has $x_i < s_x$, decrease its coordinate by s_x and decrement its period. Do not move $\|$.

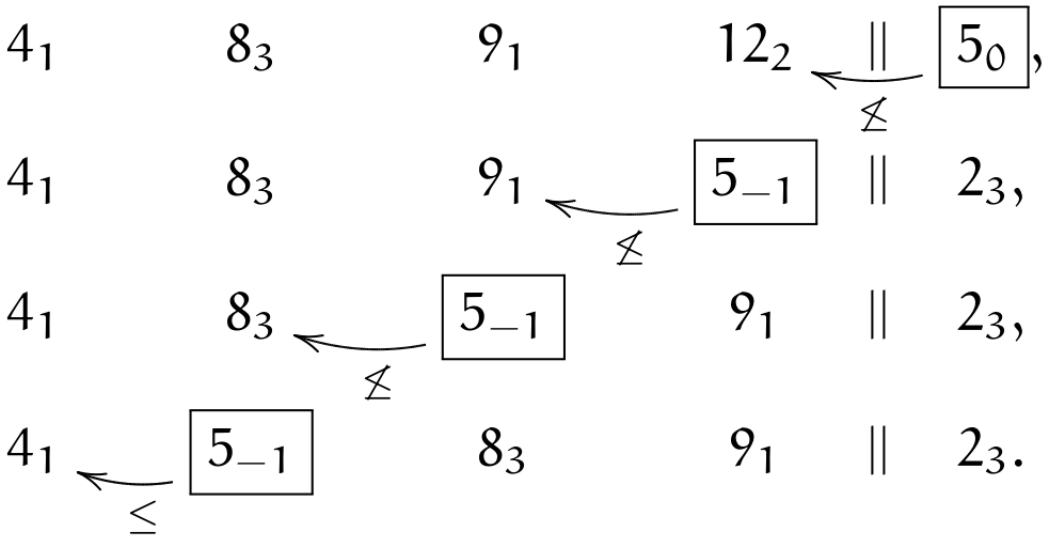
In the first step, $(\|\bullet)$ is applied, and inversion with 12_2 happens; then we stop because of $(\leq)$:



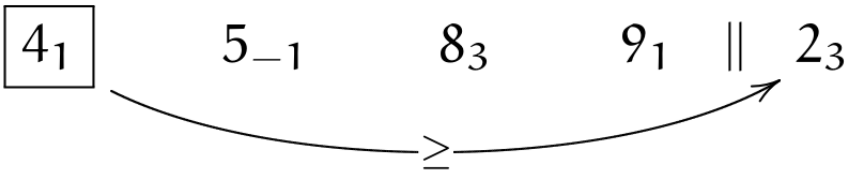
We move to next element $\boxed{-2_4}$; first, we apply ($\|\bullet$), then invert until ($\leq$):



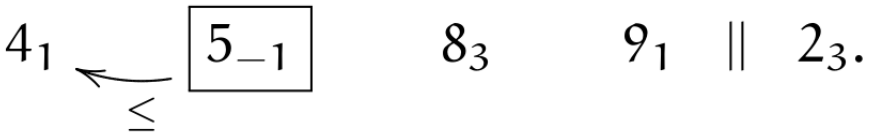
The next element is $\boxed{5_0}$; we satisfy ($\nparallel$), therefore instead of comparing $12_2 > 5_0$, we must do $(12_2 - s_x) = 2_3 \leq 5$; we adjust periods when swapping over $\|$ and apply ($\|\circ$), turning 12_2 into 2_3; then we keep inverting, until ($\leq$):



We move (wrapping around) to $\boxed{4_1}$, which is ordered:



and so is the last element



2.1.7 Computational aspects

Cost

The DEM computation using an explicit integration scheme demands a relatively high number of steps during simulation, compared to implicit scehemes. The total computation time Z of simulation spanning T seconds (of simulated time), containing N particles in volume V depends on:

- linearly, the number of steps $i = T/(s_t \Delta t_{cr})$, where s_t is timestep safety factor; Δt_{cr} can be estimated by p-wave velocity using E and ρ (sect. *Estimation of \Dtcr by wave propagation speed*) as $\Delta t_{cr}^{(p)} = r\sqrt{\frac{\rho}{E}}$. Therefore

$$i = \frac{T}{s_t r}\sqrt{\frac{E}{\rho}}.$$

- the number of particles N; for fixed value of simulated domain volume V and particle radius r

$$N = p\frac{V}{\frac{4}{3}\pi r^3},$$

  where p is packing porosity, roughly $\frac{1}{2}$ for dense irregular packings of spheres of similar radius.

  The dependency is not strictly linear (which would be the best case), as some algorithms do not scale linearly; a case in point is the sweep and prune collision detection algorithm introduced in sect. *Sweep and prune*, with scaling roughly $\mathcal{O}(N \log N)$.

  The number of interactions scales with N, as long as packing characteristics are the same.

- the number of computational cores n_{cpu}; in the ideal case, the dependency would be inverse-linear were all algorithms parallelized (in Yade, collision detection is not).

Let us suppose linear scaling. Additionally, let us suppose that the material to be simulated (E, ρ) and the simulation setup (V, T) are given in advance. Finally, dimensionless constants s_t, p and n_{cpu} will have a fixed value. This leaves us with one last degree of freedom, r. We may write

$$Z \propto iN\frac{1}{n_{cpu}} = \frac{T}{s_t r}\sqrt{\frac{E}{\rho}}p\frac{V}{\frac{4}{3}\pi r^3}\frac{1}{n_{cpu}} \propto \frac{1}{r}\frac{1}{r^3} = \frac{1}{r^4}.$$

This (rather trivial) result is essential to realize DEM scaling; if we want to have finer results, refining the "mesh" by halving r, the computation time will grow $2^4 = 16$ times.

For very crude estimates, one can use a known simulation to obtain a machine "constant"

$$\mu = \frac{Z}{Ni}$$

with the meaning of time per particle and per timestep (in the order of 10^{-6} s for current machines). μ will be only useful if simulation characteristics are similar and non-linearities in scaling do not have major influence, i.e. N should be in the same order of magnitude as in the reference case.

Result indeterminism

It is naturally expected that running the same simulation several times will give exactly the same results: although the computation is done with finite precision, round-off errors would be deterministically the same at every run. While this is true for *single-threaded* computation where exact order of all operations is given by the simulation itself, it is not true anymore in *multi-threaded* computation which is described in detail in later sections.

The straight-forward manner of parallel processing in explicit DEM is given by the possibility of treating interactions in arbitrary order. Strain and stress is evaluated for each interaction independently, but forces from interactions have to be summed up. If summation order is also arbitrary (in Yade, forces are accumulated for each thread in the order interactions are processed, then summed together), then the results can be slightly different. For instance

```
(1/10.)+(1/13.)+(1/17.)=0.23574660633484162
(1/17.)+(1/13.)+(1/10.)=0.23574660633484165
```

As forces generated by interactions are assigned to bodies in quasi-random order, summary force F_i on the body can be different between single-threaded and multi-threaded computations, but also between different runs of multi-threaded computation with exactly the same parameters. Exact thread scheduling by the kernel is not predictable since it depends on asynchronous events (hardware interrupts) and other unrelated tasks running on the system; and it is thread scheduling that ultimately determines summation order of force contributions from interactions.

2.2 User's manual

2.2.1 Scene construction

Adding particles

The *BodyContainer* holds *Body* objects in the simulation; it is accessible as `O.bodies`.

Creating Body objects

Body objects are only rarely constructed by hand by their components (*Shape*, *Bound*, *State*, *Material*); instead, convenience functions *sphere*, *facet* and *wall* are used to create them. Using these functions also ensures better future compatibility, if internals of *Body* change in some way. These functions receive geometry of the particle and several other characteristics. See their documentation for details. If the same *Material* is used for several (or many) bodies, it can be shared by adding it in `O.materials`, as explained below.

Defining materials

The `O.materials` object (instance of *Omega.materials*) holds defined shared materials for bodies. It only supports addition, and will typically hold only a few instances (though there is no limit).

`label` given to each material is optional, but can be passed to *sphere* and other functions for constructing body. The value returned by `O.materials.append` is an `id` of the material, which can be also passed to *sphere* – it is a little bit faster than using label, though not noticeable for small number of particles and perhaps less convenient.

If no *Material* is specified when calling *sphere*, the *last* defined material is used; that is a convenient default. If no material is defined yet (hence there is no last material), a default material will be created: FrictMat(density=1e3,young=1e7,poisson=.3,frictionAngle=.5). This should not happen for serious simulations, but is handy in simple scripts, where exact material properties are more or less irrelevant.

```
Yade [1]: len(O.materials)
Out[1]: 0

Yade [2]: idConcrete=O.materials.append(FrictMat(young=30e9,poisson=.2,frictionAngle=.
```

(continues on next page)

(continued from previous page)

```
↪6,label="concrete"))

Yade [3]: O.materials[idConcrete]
Out[3]: <FrictMat instance at 0x495eac0>

# uses the last defined material
Yade [4]: O.bodies.append(sphere(center=(0,0,0),radius=1))
Out[4]: 0

# material given by id
Yade [5]: O.bodies.append(sphere((0,0,2),1,material=idConcrete))
Out[5]: 1

# material given by label
Yade [6]: O.bodies.append(sphere((0,2,0),1,material="concrete"))
Out[6]: 2

Yade [7]: idSteel=O.materials.append(FrictMat(young=210e9,poisson=.25,frictionAngle=.
↪8,label="steel"))

Yade [8]: len(O.materials)
Out[8]: 2

# implicitly uses "steel" material, as it is the last one now
Yade [9]: O.bodies.append(facet([(1,0,0),(0,1,0),(-1,-1,0)]))
Out[9]: 3
```

Adding multiple particles

As shown above, bodies are added one by one or several at the same time using the `append` method:

```
Yade [10]: O.bodies.append(sphere((0,10,0),1))
Out[10]: 0

Yade [11]: O.bodies.append(sphere((0,0,2),1))
Out[11]: 1

# this is the same, but in one function call
Yade [12]: O.bodies.append([
   ....:   sphere((0,0,0),1),
   ....:   sphere((1,1,3),1)
   ....: ])
   ....:
Out[12]: [2, 3]
```

Many functions introduced in next sections return list of bodies which can be readily added to the simulation, including

- packing generators, such as *pack.randomDensePack*, *pack.regularHexa*
- surface function *pack.gtsSurface2Facets*
- import functions *ymport.gmsh*, *ymport.stl*, ...

As those functions use *sphere* and *facet* internally, they accept additional arguments passed to those functions. In particular, material for each body is selected following the rules above (last one if not specified, by label, by index, etc.).

Clumping particles together

In some cases, you might want to create rigid aggregate of individual particles (i.e. particles will retain their mutual position during simulation). This we call a *clump*. A clump is internally represented by a special *body*, referenced by *clumpId* of its members (see also *isClump*, *isClumpMember* and *isStandalone*). Like every body a clump has a *position*, which is the (mass) balance point between all members. A clump body itself has no *interactions* with other bodies. Interactions between clumps is represented by interactions between clump members. There are no interactions between clump members of the same clump.

YADE supports different ways of creating clumps:

- Create clumps and spheres (clump members) directly with one command:

The function *appendClumped()* is designed for this task. For instance, we might add 2 spheres tied together:

```
Yade [13]: O.bodies.appendClumped([
   ....:    sphere([0,0,0],1),
   ....:    sphere([0,0,2],1)
   ....: ])
   ....:
Out[13]: (2, [0, 1])

Yade [14]: len(O.bodies)
Out[14]: 3

Yade [15]: O.bodies[1].isClumpMember, O.bodies[2].clumpId
Out[15]: (True, 2)

Yade [16]: O.bodies[2].isClump, O.bodies[2].clumpId
Out[16]: (True, 2)
```

-> *appendClumped()* returns a tuple of ids `(clumpId,[memberId1,memberId2,...])`

- Use existing spheres and clump them together:

For this case the function *clump()* can be applied on a list of existing bodies:

```
Yade [17]: bodyList = []

Yade [18]: for ii in range(0,5):
   ....:    bodyList.append(O.bodies.append(sphere([ii,0,1],.5)))#create a "chain" of␣
 ↪5 spheres
   ....:

Yade [19]: print(bodyList)
[0, 1, 2, 3, 4]

Yade [20]: idClump=O.bodies.clump(bodyList)
```

-> *clump()* returns `clumpId`

- Another option is to replace *standalone* spheres from a given packing (see *SpherePack* and *makeCloud*) by clumps using clump templates.

This is done by a function called *replaceByClumps()*. This function takes a list of *clumpTemplates()* and a list of amounts and replaces spheres by clumps. The volume of a new clump will be the same as the volume of the sphere, that was replaced (clump volume/mass/inertia is accounting for overlaps assuming that there are only pair overlaps).

-> *replaceByClumps()* returns a list of tuples: `[(clumpId1,[memberId1,memberId2,...]),(clumpId2,[memberId1,memberId2,...]),...]`

It is also possible to *add* bodies to a clump and *release* bodies from a clump. Also you can *erase* the clump (clump members will become standalone).

Additionally YADE allows to achieve the *roundness* of a clump or roundness coefficient of a packing. Parts of the packing can be excluded from roundness measurement via exclude list.

```
Yade [21]: bodyList = []

Yade [22]: for ii in range(1,5):
   ....:    bodyList.append(O.bodies.append(sphere([ii,ii,ii],.5)))
   ....:

Yade [23]: O.bodies.clump(bodyList)
Out[23]: 4

Yade [24]: RC=O.bodies.getRoundness()

Yade [25]: print(RC)
0.25619141423166986
```

-> *getRoundness()* returns roundness coefficient RC of a packing or a part of the packing

Note

Have a look at examples/clumps/ folder. There you will find some examples, that show usage of different functions for clumps.

Sphere packings

Representing a solid of an arbitrary shape by arrangement of spheres presents the problem of sphere packing, i.e. spatial arrangement of spheres such that a given solid is approximately filled with them. For the purposes of DEM simulation, there can be several requirements.

1. Distribution of spheres' radii. Arbitrary volume can be filled completely with spheres provided there are no restrictions on their radius; in such case, number of spheres can be infinite and their radii approach zero. Since both number of particles and minimum sphere radius (via critical timestep) determine computation cost, radius distribution has to be given mandatorily. The most typical distribution is uniform: mean±dispersion; if dispersion is zero, all spheres will have the same radius.
2. Smooth boundary. Some algorithms treat boundaries in such way that spheres are aligned on them, making them smoother as surface.
3. Packing density, or the ratio of spheres volume and solid size. It is closely related to radius distribution.
4. Coordination number, (average) number of contacts per sphere.
5. Isotropy (related to regularity/irregularity); packings with preferred directions are usually not desirable, unless the modeled solid also has such preference.
6. Permissible Spheres' overlap; some algorithms might create packing where spheres slightly overlap; since overlap usually causes forces in DEM, overlap-free packings are sometimes called "stress-free .

Volume representation

There are 2 methods for representing exact volume of the solid in question in Yade: boundary representation and constructive solid geometry. Despite their fundamental differences, they are abstracted in Yade in the *Predicate* class. Predicate provides the following functionality:

1. defines axis-aligned bounding box for the associated solid (optionally defines oriented bounding box);

2. can decide whether given point is inside or outside the solid; most predicates can also (exactly or approximately) tell whether the point is inside *and* satisfies some given padding distance from the represented solid boundary (so that sphere of that volume doesn't stick out of the solid).

Constructive Solid Geometry (CSG)

CSG approach describes volume by geometric *primitives* or primitive solids (sphere, cylinder, box, cone, ...) and boolean operations on them. Primitives defined in Yade include *inCylinder*, *inSphere*, *inEllipsoid*, *inHyperboloid*, *notInNotch*.

For instance, *hyperboloid* (dogbone) specimen for tension-compression test can be constructed in this way (shown at img. *img-hyperboloid*):

```
from yade import pack

## construct the predicate first
pred=pack.inHyperboloid(centerBottom=(0,0,-.1),centerTop=(0,0,.1),radius=.05,skirt=.
↪03)
## alternatively: pack.inHyperboloid((0,0,-.1),(0,0,.1),.05,.03)

## pack the predicate with spheres (will be explained later)
spheres=pack.randomDensePack(pred,spheresInCell=2000,radius=3.5e-3)

## add spheres to simulation
O.bodies.append(spheres)
```

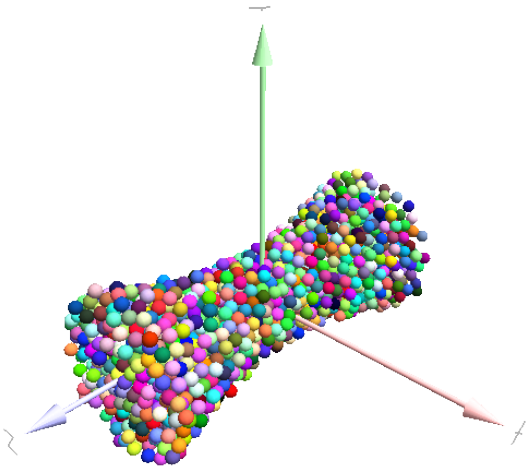


Fig. 8: Specimen constructed with the *pack.inHyperboloid* predicate, packed with *pack.randomDensePack*.

Boundary representation (BREP)

Representing a solid by its boundary is much more flexible than CSG volumes, but is mostly only approximate. Yade interfaces to GNU Triangulated Surface Library (GTS) to import surfaces readable by GTS, but also to construct them explicitly from within simulation scripts. This makes possible parametric construction of rather complicated shapes; there are functions to create set of 3d polylines from 2d polyline (*pack.revolutionSurfaceMeridians*), to triangulate surface between such set of 3d polylines (*pack.sweptPolylines2gtsSurface*).

For example, we can construct a simple funnel (examples/funnel.py, shown at *img-funnel*):

```
from numpy import linspace
from yade import pack
```

(continues on next page)

(continued from previous page)

```
# angles for points on circles
thetas=linspace(0,2*pi,num=16,endpoint=True)

# creates list of polylines in 3d from list of 2d projections
# turned from 0 to
meridians=pack.revolutionSurfaceMeridians(
        [[(3+rad*sin(th),10*rad+rad*cos(th)) for th in thetas] for rad in linspace(1,
↪2,num=10)],
        linspace(0,pi,num=10)
)

# create surface
surf=pack.sweptPolylines2gtsSurface(
        meridians
        +[[Vector3(5*sin(-th),-10+5*cos(-th),30) for th in thetas]]  # add funnel top
)

# add to simulation
O.bodies.append(pack.gtsSurface2Facets(surf))
```

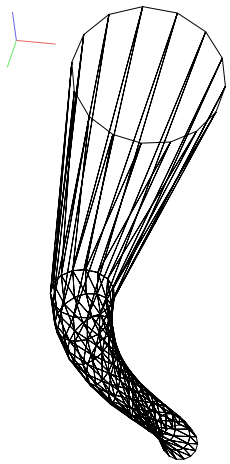


Fig. 9: Triangulated funnel, constructed with the examples/funnel.py script.

GTS surface objects can be used for 2 things:

1. *pack.gtsSurface2Facets* function can create the triangulated surface (from *Facet* particles) in the simulation itself, as shown in the funnel example. (Triangulated surface can also be imported directly from a STL file using *ymport.stl*.)
2. *pack.inGtsSurface* predicate can be created, using the surface as boundary representation of the enclosed volume.

The examples/gts-horse/gts-horse.py (img. *img-horse*) shows both possibilities; first, a GTS surface is imported:

```
import gts
surf=gts.read(open('horse.coarse.gts'))
```

That surface object is used as predicate for packing:

```
pred=pack.inGtsSurface(surf)
aabb=pred.aabb()
```

(continues on next page)

(continued from previous page)

```
radius=(aabb[1][0]-aabb[0][0])/40
O.bodies.append(pack.regularHexa(pred,radius=radius,gap=radius/4.))
```

and then, after being translated, as base for triangulated surface in the simulation itself:

```
surf.translate(0,0,-(aabb[1][2]-aabb[0][2]))
O.bodies.append(pack.gtsSurface2Facets(surf,wire=True))
```

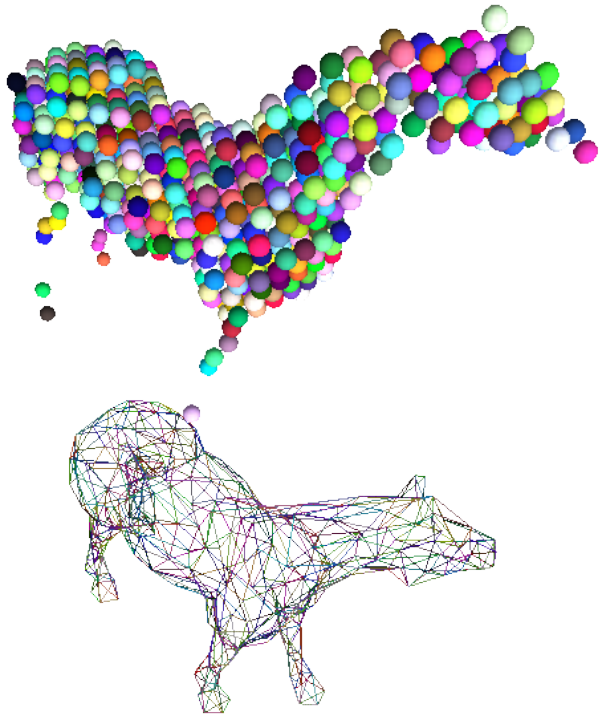


Fig. 10: Imported GTS surface (horse) used as packing predicate (top) and surface constructed from *facets* (bottom). See http://www.youtube.com/watch?v=PZVrulIUX1A for movie of this simulation.

Boolean operations on predicates

Boolean operations on pair of predicates (noted `A` and `B`) are defined:

- *intersection* `A & B` (conjunction): point must be in both predicates involved.
- *union* `A | B` (disjunction): point must be in the first or in the second predicate.
- *difference* `A - B` (conjunction with second predicate negated): the point must be in the first predicate and not in the second one.
- *symmetric difference* `A ^ B` (exclusive disjunction): point must be in exactly one of the two predicates.

Composed predicates also properly define their bounding box. For example, we can take box and remove cylinder from inside, using the `A - B` operation (img. *img-predicate-difference*):

```
pred=pack.inAlignedBox((-2,-2,-2),(2,2,2))-pack.inCylinder((0,-2,0),(0,2,0),1)
spheres=pack.randomDensePack(pred,spheresInCell=2000,radius=.1,rRelFuzz=.4,
↪returnSpherePack=True)
spheres.toSimulation()
```

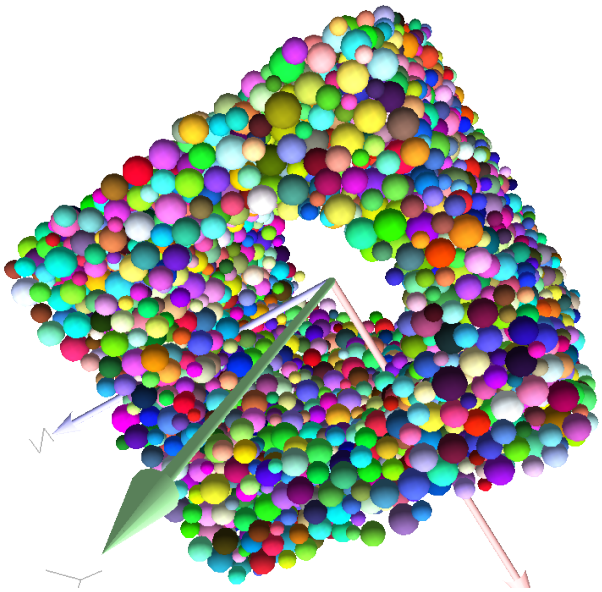


Fig. 11: Box with cylinder removed from inside, using difference of these two predicates.

Packing algorithms

Algorithms presented below operate on geometric spheres, defined by their center and radius. With a few exception documented below, the procedure is as follows:

1. Sphere positions and radii are computed (some functions use volume predicate for this, some do not)
2. *sphere* is called for each position and radius computed; it receives extra keyword arguments of the packing function (i.e. arguments that the packing function doesn't specify in its definition; they are noted `**kw`). Each *sphere* call creates actual *Body* objects with *Sphere shape*. List of *Body* objects is returned.
3. List returned from the packing function can be added to simulation using *toSimulation()*. Legacy code used a call to *O.bodies.append*.

Taking the example of pierced box:

```
pred=pack.inAlignedBox((-2,-2,-2),(2,2,2))-pack.inCylinder((0,-2,0),(0,2,0),1)
spheres=pack.randomDensePack(pred,spheresInCell=2000,radius=.1,rRelFuzz=.4,wire=True,
↪color=(0,0,1),material=1,returnSpherePack=True)
```

Keyword arguments `wire`, `color` and `material` are not declared in *pack.randomDensePack*, therefore will be passed to *sphere*, where they are also documented. `spheres` is now a *SpherePack* object.:

```
spheres.toSimulation()
```

Packing algorithms described below produce dense packings. If one needs loose packing, *SpherePack* class provides functions for generating loose packing, via its *makeCloud()* method. It is used internally for generating initial configuration in dynamic algorithms. For instance:

```
from yade import pack
sp=pack.SpherePack()
sp.makeCloud(minCorner=(0,0,0),maxCorner=(3,3,3),rMean=.2,rRelFuzz=.5)
```

will fill given box with spheres, until no more spheres can be placed. The object can be used to add spheres to simulation:

```
sp.toSimulation()
```

Geometric

Geometric algorithms compute packing without performing dynamic simulation; among their advantages are

- speed;
- spheres touch exactly, there are no overlaps (what some people call "stress-free" packing);

their chief disadvantage is that radius distribution cannot be prescribed exactly, save in specific cases (regular packings); sphere radii are given by the algorithm, which already makes the system determined. If exact radius distribution is important for your problem, consider dynamic algorithms instead.

Regular

Yade defines packing generators for spheres with constant radii, which can be used with volume predicates as described above. They are dense orthogonal packing (*pack.regularOrtho*) and dense hexagonal packing (*pack.regularHexa*). The latter creates so-called "hexagonal close packing", which achieves maximum density (http://en.wikipedia.org/wiki/Close-packing_of_spheres).

Clear disadvantage of regular packings is that they have very strong directional preferences, which might not be an issue in some cases.

Irregular

Random geometric algorithms do not integrate at all with volume predicates described above; rather, they take their own boundary/volume definition, which is used during sphere positioning. On the other hand, this makes it possible for them to respect boundary in the sense of making spheres touch it at appropriate places, rather than leaving empty space in-between.

GenGeo
: is library (python module) for packing generation developed with ESyS-Particle. It creates packing by random insertion of spheres with given radius range. Inserted spheres touch each other exactly and, more importantly, they also touch the boundary, if in its neighbourhood. Boundary is represented as special object of the GenGeo library (Sphere, cylinder, box, convex polyhedron, ...). Therefore, GenGeo cannot be used with volume represented by yade predicates as explained above.

  Packings generated by this module can be imported directly via *ymport.gengeo*, or from saved file via *ymport.gengeoFile*. There is an example script examples/test/genCylLSM.py. Full documentation for GenGeo can be found at ESyS documentation website.

  There are debian packages esys-particle and python-demgengeo.

Dynamic

The most versatile algorithm for random dense packing is provided by *pack.randomDensePack*. Initial loose packing of non-overlapping spheres is generated by randomly placing them in cuboid volume, with radii given by requested (currently only uniform) radius distribution. When no more spheres can be inserted, the packing is compressed and then uncompressed (see py/pack/pack.py for exact values of these "stresses") by running a DEM simulation; *Omega.switchScene* is used to not affect existing simulation). Finally, resulting packing is clipped using provided predicate, as explained above.

By its nature, this method might take relatively long; and there are 2 provisions to make the computation time shorter:

- If number of spheres using the `spheresInCell` parameter is specified, only smaller specimen with *periodic* boundary is created and then repeated as to fill the predicate. This can provide high-quality packing with low regularity, depending on the `spheresInCell` parameter (value of several thousands is recommended).

- Providing `memoizeDb` parameter will make *pack.randomDensePack* first look into provided file (SQLite database) for packings with similar parameters. On success, the packing is simply read from database and returned. If there is no similar pre-existent packing, normal procedure is run, and the result is saved in the database before being returned, so that subsequent calls with same parameters will return quickly.

If you need to obtain full periodic packing (rather than packing clipped by predicate), you can use *pack.randomPeriPack*.

In case of specific needs, you can create packing yourself, "by hand". For instance, packing boundary can be constructed from *facets*, letting randomly positioned spheres in space fall down under gravity.

Triangulated surfaces

Yade integrates with the the GNU Triangulated Surface library, exposed in python via GTS module. GTS provides variety of functions for surface manipulation (coarsening, tesselation, simplification, import), to be found in its documentation.

GTS surfaces are geometrical objects, which can be inserted into simulation as set of particles whose *Body.shape* is of type *Facet* – single triangulation elements. *pack.gtsSurface2Facets* can be used to convert GTS surface triangulation into list of *bodies* ready to be inserted into simulation via `O.bodies.append`.

Facet particles are created by default as non-*Body.dynamic* (they have zero inertial mass). That means that they are fixed in space and will not move if subject to forces. You can however

- prescribe arbitrary movement to facets using a *PartialEngine* (such as *TranslationEngine* or *RotationEngine*);
- assign explicitly *mass* and *inertia* to that particle;
- make that particle part of a clump and assign *mass* and *inertia* of the clump itself (described below).

Note

Facets can only (currently) interact with *spheres*, not with other facets, even if they are *dynamic*. Collision of 2 *facets* will not create interaction, therefore no forces on facets.

Import

Yade currently offers 3 formats for importing triangulated surfaces from external files, in the *ymport* module:

ymport.gts
: text file in native GTS format.

ymport.stl
: STereoLitography format, in either text or binary form; exported from Blender, but from many CAD systems as well.

***ymport.gmsh*.**
: text file in native format for GMSH, popular open-source meshing program.

If you need to manipulate surfaces before creating list of facets, you can study the py/ymport.py file where the import functions are defined. They are rather simple in most cases.

Parametric construction

The GTS module provides convenient way of creating surface by vertices, edges and triangles.

Frequently, though, the surface can be conveniently described as surface between polylines in space. For instance, cylinder is surface between two polygons (closed polylines). The *pack.sweptPolylines2gtsSurface* offers the functionality of connecting several polylines with triangulation.

Note

The implementation of *pack.sweptPolylines2gtsSurface* is rather simplistic: all polylines must be of the same length, and they are connected with triangles between points following their indices within each polyline (not by distance). On the other hand, points can be co-incident, if the `threshold` parameter is positive: degenerate triangles with vertices closer that `threshold` are automatically eliminated.

Manipulating lists efficiently (in terms of code length) requires being familiar with list comprehensions in python.

Another examples can be found in examples/mill.py (fully parametrized) or examples/funnel.py (with hardcoded numbers).

Creating interactions

In typical cases, interactions are created during simulations as particles collide. This is done by a *Collider* detecting approximate contact between particles and then an *IGeomFunctor* detecting exact collision.

Some material models (such as the *concrete model*) rely on initial interaction network which is denser than geometrical contact of spheres: sphere's radii as "enlarged" by a dimensionless factor called *interaction radius* (or *interaction ratio*) to create this initial network. This is done typically in this way (see examples/concrete/uniax.py for an example):

1. Approximate collision detection is adjusted so that approximate contacts are detected also between particles within the interaction radius. This consists in setting value of *Bo1_Sphere_Aabb.aabbEnlargeFactor* to the interaction radius value.
2. The geometry functor (`Ig2`) would normally say that "there is no contact" if given 2 spheres that are not in contact. Therefore, the same value as for *Bo1_Sphere_Aabb.aabbEnlargeFactor* must be given to it (*Ig2_Sphere_Sphere_ScGeom.interactionDetectionFactor*).

   Note that only *Sphere* + *Sphere* interactions are supported; there is no parameter analogous to *distFactor* in *Ig2_Facet_Sphere_ScGeom*. This is on purpose, since the interaction radius is meaningful in bulk material represented by sphere packing, whereas facets usually represent boundary conditions which should be exempt from this dense interaction network.
3. Run one single step of the simulation so that the initial network is created.
4. Reset interaction radius in both `Bo1` and `Ig2` functors to their default value again.
5. Continue the simulation; interactions that are already established will not be deleted (the `Law2` functor in use permitting).

In code, such scenario might look similar to this one (labeling is explained in *Labeling things*):

```
intRadius=1.5
damping=0.05

O.engines=[
   ForceResetter(),
   InsertionSortCollider([
      # enlarge here
      Bo1_Sphere_Aabb(aabbEnlargeFactor=intRadius,label='bo1s'),
      Bo1_Facet_Aabb(),
        ]),
   InteractionLoop(
      [
         # enlarge here
         Ig2_Sphere_Sphere_ScGeom(interactionDetectionFactor=intRadius,label='ig2ss'),
         Ig2_Facet_Sphere_ScGeom(),
```

(continues on next page)

(continued from previous page)

```
      ],
      [Ip2_CpmMat_CpmMat_CpmPhys()],
      [Law2_ScGeom_CpmPhys_Cpm(epsSoft=0)], # deactivated
   ),
   NewtonIntegrator(damping=damping,label='damper'),
]

# run one single step
O.step()

# reset interaction radius to the default value
bo1s.aabbEnlargeFactor=1.0
ig2ss.interactionDetectionFactor=1.0

# now continue simulation
O.run()
```

Individual interactions on demand

It is possible to create an interaction between a pair of particles independently of collision detection using *createInteraction*. This function looks for and uses matching `Ig2` and `Ip2` functors. Interaction will be created regardless of distance between given particles (by passing a special parameter to the `Ig2` functor to force creation of the interaction even without any geometrical contact). Appropriate constitutive law should be used to avoid deletion of the interaction at the next simulation step.

```
Yade [26]: O.materials.append(FrictMat(young=3e10,poisson=.2,density=1000))
Out[26]: 0

Yade [27]: O.bodies.append([
   ....:    sphere([0,0,0],1),
   ....:    sphere([0,0,1000],1)
   ....: ])
   ....:
Out[27]: [0, 1]

# only add InteractionLoop, no other engines are needed now
Yade [28]: O.engines=[
   ....:    InteractionLoop(
   ....:        [Ig2_Sphere_Sphere_ScGeom(),],
   ....:        [Ip2_FrictMat_FrictMat_FrictPhys()],
   ....:        [] # not needed now
   ....:    )
   ....: ]
   ....:

Yade [29]: i=createInteraction(0,1)

# created by functors in InteractionLoop
Yade [30]: i.geom, i.phys
Out[30]: (<ScGeom instance at 0x494e9b0>, <FrictPhys instance at 0x4bb08a0>)
```

This method will be rather slow if many interactions are to be created (the functor lookup will be repeated for each of them). In such case, ask on gitlab answers to have the *createInteraction* function accept list of pairs id's as well.

Assigning cohesive bonds (possibly between distant bodies)

A particular case of interactions on demand is when cohesive strength needs to be assigned to interactions when using *Law2_ScGeom6D_CohFrictPhys_CohesionMoment*.

- For existing interactions, it can be done using *setCohesion*. If *physFunctor* is physics functor of type *Ip2_CohFrictMat_CohFrictMat_CohFrictPhys* and *i* an existing, cohesionless, interaction, this line would assign cohesion without changing the current contact force/moments:

```
physFunctor.setCohesion(i,cohesive=True,resetDisp=False)
```

- For creating new interactions (particularly when the interaction needs to be created between distant bodies), the interaction needs to be created, first, then cohesion can be assigned:

```
i=createInteraction(i.id1,i.id2)
physFunctor.setCohesion(i,cohesive=True,resetDisp=False)
```

If the above lines are executed between distant bodies, the interaction will be initially in traction. In order to make it force free we need to use current distance as equilibrium distance, this is achieved with *resetDisp=True*.

- Creating cohesive bonds with this method between many thousand of distant bodies can be a challenge since it needs to identify the candidate pairs. In such situation, it is suggested to exploit the collision detection engine to establish the list of close - though distant - neighbours. Indeed, the collider can be executed outside the time-integration loop, with a user-defined detection distance, in order to produce that list. This technique is examplified in examples/cohesion/assignCohesionRemote.py.

Base engines

A typical DEM simulation in Yade does at least the following at each step (see *Function components* for details):

1. Reset forces from previous step
2. Detect new collisions
3. Handle interactions
4. Apply forces and update positions of particles

Each of these points corresponds to one or several engines:

```
O.engines=[
   ForceResetter(),          # reset forces
   InsertionSortCollider([...]),  # approximate collision detection
   InteractionLoop([...],[...],[...]) # handle interactions
   NewtonIntegrator()        # apply forces and update positions
]
```

The order of engines is important. In majority of cases, you will put any additional engine after *InteractionLoop*:

- if it applies force, it should come before *NewtonIntegrator*, otherwise the force will never be effective.
- if it makes use of bodies' positions, it should also come before *NewtonIntegrator*, otherwise, positions at the next step will be used (this might not be critical in many cases, such as output for visualization with *VTKRecorder*).

The *O.engines* sequence must be always assigned at once (the reason is in the fact that although engines themselves are passed by reference, the sequence is *copied* from c++ to Python or from Python to c++). This includes modifying an existing `O.engines`; therefore

```
O.engines.append(SomeEngine()) # wrong
```

will not work;

```
O.engines=O.engines+[SomeEngine()] # ok
```

must be used instead. For inserting an engine after position #2 (for example), use python slice notation:

```
O.engines=O.engines[:2]+[SomeEngine()]+O.engines[2:]
```

Note

When Yade starts, O.engines is filled with a reasonable default list, so that it is not strictly necessary to redefine it when trying simple things. The default scene will handle spheres, boxes, and facets with *frictional* properties correctly, and adjusts the timestep dynamically. You can find an example in examples/simple-scene/simple-scene-default-engines.py.

Functors choice

In the above example, we omited functors, only writing ellipses ... instead. As explained in *Dispatchers and functors*, there are 4 kinds of functors and associated dispatchers. User can choose which ones to use, though the choice must be consistent.

Bo1 functors

`Bo1` functors must be chosen depending on the collider in use; they are given directly to the collider (which internally uses *BoundDispatcher*).

At this moment (January 2019), the most common choice is *InsertionSortCollider*, which uses *Aabb*; functors creating *Aabb* must be used in that case. Depending on particle *shapes* in your simulation, choose appropriate functors:

```
O.engines=[...,
   InsertionSortCollider([Bo1_Sphere_Aabb(),Bo1_Facet_Aabb()]),
   ...
]
```

Using more functors than necessary (such as *Bo1_Facet_Aabb* if there are no *facets* in the simulation) has no performance penalty. On the other hand, missing functors for existing *shapes* will cause those bodies to not collide with other bodies (they will freely interpenetrate).

There are other *colliders* as well, though their usage is only experimental:

- *SpatialQuickSortCollider* is correctness-reference collider operating on *Aabb*; it is significantly slower than *InsertionSortCollider*.
- *PersistentTriangulationCollider* only works on spheres; it does not use a *BoundDispatcher*, as it operates on spheres directly.
- *FlatGridCollider* is proof-of-concept grid-based collider, which computes grid positions internally (no *BoundDispatcher* either)

Ig2 functors

`Ig2` functor choice (all of them derive from *IGeomFunctor*) depends on

1. shape combinations that should collide; for instance:

   ```
   InteractionLoop([Ig2_Sphere_Sphere_ScGeom()],[],[])
   ```

will handle collisions for *Sphere* + *Sphere*, but not for *Facet* + *Sphere* – if that is desired, an additional functor must be used:

```
InteractionLoop([
   Ig2_Sphere_Sphere_ScGeom(),
   Ig2_Facet_Sphere_ScGeom()
],[],[])
```

Again, missing combination will cause given shape combinations to freely interpenetrate one another. There are several possible choices of a functor for each pair, hence they cannot be put into *InsertionSortCollider* by default. A common mistake for bodies going through each other is that the necessary functor was not added.

2. *IGeom* type accepted by the `Law2` functor (below); it is the first part of functor's name after `Law2` (for instance, *Law2_ScGeom_CpmPhys_Cpm* accepts *ScGeom*).

Ip2 functors

`Ip2` functors (deriving from *IPhysFunctor*) must be chosen depending on

1. *Material* combinations within the simulation. In most cases, `Ip2` functors handle 2 instances of the same *Material* class (such as *Ip2_FrictMat_FrictMat_FrictPhys* for 2 bodies with *FrictMat*)
2. *IPhys* accepted by the constitutive law (`Law2` functor), which is the second part of the `Law2` functor's name (e.g. *Law2_ScGeom_FrictPhys_CundallStrack* accepts *FrictPhys*)

Note

Unlike with `Bo1` and `Ig2` functors, unhandled combination of *Materials* is an error condition signaled by an exception.

Law2 functor(s)

`Law2` functor was the ultimate criterion for the choice of `Ig2` and `Ip2` functors; there are no restrictions on its choice in itself, as it only applies forces without creating new objects.

In most simulations, only one `Law2` functor will be in use; it is possible, though, to have several of them, dispatched based on combination of *IGeom* and *IPhys* produced previously by `Ig2` and `Ip2` functors respectively (in turn based on combination of *Shapes* and *Materials*).

Note

As in the case of `Ip2` functors, receiving a combination of *IGeom* and *IPhys* which is not handled by any `Law2` functor is an error.

Warning

Many `Law2` exist in Yade, and new ones can appear at any time. In some cases different functors are only different implementations of the same contact law (e.g. *Law2_ScGeom_FrictPhys_CundallStrack* and *Law2_L3Geom_FrictPhys_ElPerfPl*). Also, sometimes, the peculiarity of one functor may be reproduced as a special case of a more general one. Therefore, for a given constitutive behavior, the user may have the choice between different functors. It is strongly recommended to favor the most used and most validated implementation when facing such choice. A list of available functors classified from mature to unmaintained is updated here to guide this choice.

Examples

Let us give several examples of the chain of created and accepted types.

Basic DEM model

Suppose we want to use the *Law2_ScGeom_FrictPhys_CundallStrack* constitutive law. We see that

1. the `Ig2` functors must create *ScGeom*. If we have for instance *spheres* and *boxes* in the simulation, we will need functors accepting *Sphere* + *Sphere* and *Box* + *Sphere* combinations. We don't want interactions between boxes themselves (as a matter of fact, there is no such functor anyway). That gives us *Ig2_Sphere_Sphere_ScGeom* and *Ig2_Box_Sphere_ScGeom*.
2. the `Ip2` functors should create *FrictPhys*. Looking at *InteractionPhysicsFunctors*, there is only *Ip2_FrictMat_FrictMat_FrictPhys*. That obliges us to use *FrictMat* for particles.

The result will be therefore:

```
InteractionLoop(
   [Ig2_Sphere_Sphere_ScGeom(),Ig2_Box_Sphere_ScGeom()],
   [Ip2_FrictMat_FrictMat_FrictPhys()],
   [Law2_ScGeom_FrictPhys_CundallStrack()]
)
```

Concrete model

In this case, our goal is to use the *Law2_ScGeom_CpmPhys_Cpm* constitutive law.

- We use *spheres* and *facets* in the simulation, which selects `Ig2` functors accepting those types and producing *ScGeom*: *Ig2_Sphere_Sphere_ScGeom* and *Ig2_Facet_Sphere_ScGeom*.
- We have to use *Material* which can be used for creating *CpmPhys*. We find that *CpmPhys* is only created by *Ip2_CpmMat_CpmMat_CpmPhys*, which determines the choice of *CpmMat* for all particles.

Therefore, we will use:

```
InteractionLoop(
   [Ig2_Sphere_Sphere_ScGeom(),Ig2_Facet_Sphere_ScGeom()],
   [Ip2_CpmMat_CpmMat_CpmPhys()],
   [Law2_ScGeom_CpmPhys_Cpm()]
)
```

Imposing conditions

In most simulations, it is not desired that all particles float freely in space. There are several ways of imposing boundary conditions that block movement of all or some particles with regard to global space.

Note

When using *Clump* bodies discussed in above section *Clumping particles together*, the following paragraphs apply to the Clump bodies themselves (not to their members).

Motion constraints

- *Body.dynamic* determines whether a body will be accelerated by *NewtonIntegrator*; it is mandatory to make it false for bodies with zero mass, where applying non-zero force would result in infinite displacement.

Facets are case in the point: *facet* makes them non-dynamic by default, as they have zero volume and zero mass (this can be changed, by passing `dynamic=True` to *facet* or setting *Body.dynamic*; setting *State.mass* to a non-zero value must be done as well). The same is true for *wall*.

Making sphere non-dynamic is achieved simply by:

```
b = sphere([x,y,z],radius,dynamic=False)
b.dynamic=True #revert the previous
```

- *State.blockedDOFs* permits selective blocking of any of 6 degrees of freedom in global space. For instance, a sphere can be made to move only in the xy plane by saying:

```
Yade [31]: O.bodies.append(sphere((0,0,0),1))
Out[31]: 0

Yade [32]: O.bodies[0].state.blockedDOFs='zXY'
```

In contrast to *Body.dynamic*, *blockedDOFs* will only block forces (and acceleration) in selected directions. Actually, `b.dynamic=False` is nearly only a shorthand for `b.state.blockedDOFs=='xyzXYZ'` . A subtle difference is that the former does reset the velocity components automaticaly, while the latest does not. If you prescribed linear or angular velocity, they will be applied regardless of *blockedDOFs*. It also implies that if the velocity is not zero when degrees of freedom are blocked via blockedDOFs assignements, the body will keep moving at the velocity it has at the time of blocking. The differences are shown below:

```
Yade [33]: b1 = sphere([0,0,0],1,dynamic=True)

Yade [34]: b1.state.blockedDOFs
Out[34]: ''

Yade [35]: b1.state.vel = Vector3(1,0,0) #we want it to move...

Yade [36]: b1.dynamic = False #... at a constant velocity

Yade [37]: print(b1.state.blockedDOFs, b1.state.vel)
xyzXYZ Vector3(0,0,0)

Yade [38]: # oops, velocity has been reset when setting dynamic=False

Yade [39]: b1.state.vel = (1,0,0) # we can still assign it now

Yade [40]: print(b1.state.blockedDOFs, b1.state.vel)
xyzXYZ Vector3(1,0,0)

Yade [41]: b2 = sphere([0,0,0],1,dynamic=True) #another try

Yade [42]: b2.state.vel = (1,0,0)

Yade [43]: b2.state.blockedDOFs = "xyzXYZ" #this time we assign blockedDOFs␣
↪directly, velocity is unchanged

Yade [44]: print(b2.state.blockedDOFs, b2.state.vel)
xyzXYZ Vector3(1,0,0)
```

It might be desirable to constrain motion of some particles constructed from a generated sphere packing, following some condition, such as being at the bottom of a specimen; this can be done by looping over all bodies with a conditional:

```
for b in O.bodies:
   # block all particles with z coord below .5:
   if b.state.pos[2]<.5: b.dynamic=False
```

Arbitrary spatial predicates introduced above can be expoited here as well:

```
from yade import pack
pred=pack.inAlignedBox(lowerCorner,upperCorner)
for b in O.bodies:
   if not isinstance(b.shape,Sphere): continue # skip non-spheres
   # ask the predicate if we are inside
   if pred(b.state.pos,b.shape.radius): b.dynamic=False
```

Imposing motion and forces

Imposed velocity

If a degree of freedom is blocked and a velocity is assigned along that direction (translational or rotational velocity), then the body will move at constant velocity. This is the simpler and recommended method to impose the motion of a body. This, for instance, will result in a constant velocity along `x` (it can still be freely accelerated along `y` and `z`):

```
O.bodies.append(sphere((0,0,0),1))
O.bodies[0].state.blockedDOFs='x'
O.bodies[0].state.vel=(10,0,0)
```

Conversely, modifying the position directly is likely to break Yade's algorithms, especially those related to collision detection and contact laws, as they are based on bodies velocities. Therefore, unless you really know what you are doing, don't do that for imposing a motion:

```
O.bodies.append(sphere((0,0,0),1))
O.bodies[0].state.blockedDOFs='x'
O.bodies[0].state.pos=10*O.dt #REALLY BAD! Don't assign position
```

Initial (angular) velocity

The above assignment of linear or angular velocities may also serve as initial conditions for *dynamic* ones, where extra care has to be taken for *aspherical* bodies, depending on the choice of the corresponding *integration algorithm*. Because of the algorithm internals described at *Orientation (aspherical)*, an initial *angular momentum* (function of the desired initial angular velocity and of the *inertia tensor*) has to be assigned instead of an initial angular velocity (that would have no effect) when using the 'Fincham1992' integration algorithm. On the other hand, angular velocities are to be directly initialized with the other two algorithms: 'delValle2023' and 'Omelyan1999'.

Imposed force

Applying a force or a torque on a body is done via functions of the *ForceContainer*. It is as simple as this:

```
O.forces.addF(0,(1,0,0)) #applies for one step
```

This way, the force applies for one time step only, and is resetted at the beginning of each step. For this reason, imposing a force at the begining of one step will have no effect at all, since it will be immediatly resetted. The only way is to place a *PyRunner* inside the simulation loop.

Applying the force permanently is possible with another function (in this case it does not matter if the command comes at the begining of the time step):

```
O.forces.setPermF(0,(1,0,0)) #applies permanently
```

The force will persist across iterations, until it is overwritten by another call to `O.forces.setPermF(id, f)` or erased by `O.forces.reset(resetAll=True)`. The permanent force on a body can be checked with `O.forces.permF(id)`.

Boundary controllers

Engines deriving from *BoundaryController* impose boundary conditions during simulation, either directly, or by influencing several bodies. You are referred to their individual documentation for details, though you might find interesting in particular

- *UniaxialStrainer* for applying strain along one axis at constant rate; useful for plotting strain-stress diagrams for uniaxial loading case. See examples/concrete/uniax.py for an example.
- *TriaxialStressController* which applies prescribed stress/strain along 3 perpendicular axes on cuboid-shaped packing using 6 walls (*Box* objects)
- *PeriTriaxController* for applying stress/strain along 3 axes independently, for simulations using periodic boundary conditions (*Cell*)

Field appliers

Engines deriving from *FieldApplier* are acting on all particles. The one most used is *GravityEngine* applying uniform acceleration field (*GravityEngine* is deprecated, use *NewtonIntegrator.gravity* instead).

Partial engines

Engines deriving from *PartialEngine* define the *ids* attribute determining bodies which will be affected. Several of them warrant explicit mention here:

- *TranslationEngine* and *RotationEngine* for applying constant speed linear and rotational motion on subscribers.
- *ForceEngine* and *TorqueEngine* applying given values of force/torque on subscribed bodies at every step.
- *StepDisplacer* for applying generalized displacement delta at every timestep; designed for precise control of motion when testing constitutive laws on 2 particles.

The real value of partial engines is when you need to prescribe a complex type of force or displacement field. For moving a body at constant velocity or for imposing a single force, the methods explained in *Imposing motion and forces* are much simpler. There are several interpolating engines (*InterpolatingDirectedForceEngine* for applying force with varying magnitude, *InterpolatingHelixEngine* for applying spiral displacement with varying angular velocity; see examples/test/helix.py and possibly others); writing a new interpolating engine is rather simple using examples of those that already exist.

Convenience features

Labeling things

Engines and functors can define a `label` attribute. Whenever the `O.engines` sequence is modified, python variables of those names are created/updated; since it happens in the `__builtins__` namespaces, these names are immediately accessible from anywhere. This was used in *Creating interactions* to change interaction radius in multiple functors at once.

Warning

Make sure you do not use label that will overwrite (or shadow) an object that you already use under that variable name. Take care not to use syntactically wrong names, such as “er*452” or “my engine”; only variable names permissible in Python can be used.

Simulation tags

Omega.tags is a dictionary (it behaves like a dictionary, although the implementation in C++ is different) mapping keys to labels. Contrary to regular python dictionaries that you could create,

- `O.tags` is *saved and loaded with simulation*;
- `O.tags` has some values pre-initialized.

After Yade startup, `O.tags` contains the following:

```
Yade [45]: dict(O.tags) # convert to real dictionary
Out[45]:
{'author': 'root~(root@yade-ci)',
 'isoTime': '20261020T045807',
 'id': '20261020T045807p3349',
 'd.id': '20261020T045807p3349',
 'id.d': '20261020T045807p3349'}
```

author
: Real name, username and machine as obtained from your system at simulation creation

id
: Unique identifier of this Yade instance (or of the instance which created a loaded simulation). It is composed of date, time and process number. Useful if you run simulations in parallel and want to avoid overwriting each other's outputs; embed `O.tags['id']` in output filenames (either as directory name, or as part of the file's name itself) to avoid it. This is explained in *Separating output files from jobs* in detail.

isoTime
: Time when simulation was created (with second resolution).

d.id, id.d
: Simulation description and id joined by period (and vice-versa). Description is used in batch jobs; in non-batch jobs, these tags are identical to id.

You can add your own tags by simply assigning value, with the restriction that the left-hand side object must be a string and must not contain `=`.

```
Yade [46]: O.tags['anythingThat I lik3']='whatever'

Yade [47]: O.tags['anythingThat I lik3']
Out[47]: 'whatever'
```

Saving python variables

Python variable lifetime is limited; in particular, if you save simulation, variables will be lost after reloading. Yade provides limited support for data persistence for this reason (internally, it uses special values of `O.tags`). The functions in question are *saveVars* and *loadVars*.

saveVars takes dictionary (variable names and their values) and a *mark* (identification string for the variable set); it saves the dictionary inside the simulation. These variables can be re-created (after the simulation was loaded from a XML file, for instance) in the `yade.params.`*mark* namespace by calling *loadVars* with the same identification *mark*:

```
Yade [48]: a=45; b=pi/3

Yade [49]: saveVars('ab',a=a,b=b)

# save simulation (we could save to disk just as well)
Yade [49]: O.saveTmp()
```

(continues on next page)

(continued from previous page)

```
Yade [51]: O.loadTmp()

Yade [52]: loadVars('ab')

Yade [53]: yade.params.ab.a
Out[53]: 45

# import like this
Yade [54]: from yade.params import ab

Yade [55]: ab.a, ab.b
Out[55]: (45, 1.0471975511965976)

# also possible
Yade [56]: from yade.params import *

Yade [57]: ab.a, ab.b
Out[57]: (45, 1.0471975511965976)
```

Enumeration of variables can be tedious if they are many; creating local scope (which is a function definition in Python, for instance) can help:

```
def setGeomVars():
        radius=4
        thickness=22
        p_t=4/3*pi
        dim=Vector3(1.23,2.2,3)
        #
        # define as much as you want here
        # it all appears in locals() (and nothing else does)
        #
        saveVars('geom',loadNow=True,**locals())

setGeomVars()
from yade.params.geom import *
# use the variables now
```

Note

Only types that can be pickled can be passed to *saveVars*.

2.2.2 Controlling simulation

Tracking variables

Running python code

A special engine *PyRunner* can be used to periodically call python code, specified via the `command` parameter. Periodicity can be controlled by specifying computation time (`realPeriod`), virtual time (`virtPeriod`) or iteration number (`iterPeriod`).

For instance, to print kinetic energy (using *kineticEnergy*) every 5 seconds, the following engine will be put to `O.engines`:

```
PyRunner(command="print('kinetic energy',kineticEnergy())",realPeriod=5)
```

For running more complex commands, it is convenient to define an external function and only call it

from within the engine. Since the `command` is run in the script's namespace, functions defined within scripts can be called. Let us print information on interaction between bodies 0 and 1 periodically:

```
def intrInfo(id1,id2):
        try:
                i=O.interactions[id1,id2]
                # assuming it is a CpmPhys instance
                print (d1,id2,i.phys.sigmaN)
        except:
                # in case the interaction doesn't exist (yet?)
                print("No interaction between",id1,id2)
O.engines=[...,
        PyRunner(command="intrInfo(0,1)",realPeriod=5)
]
```

Warning

If a function was declared inside a *live* yade session (ipython) then an error `NameError: name 'intrInfo' is not defined` will occur unless python globals() are updated with command

```
globals().update(locals())
```

More useful examples will be given below.

The *plot* module provides simple interface and storage for tracking various data. Although originally conceived for plotting only, it is widely used for tracking variables in general.

The data are in *plot.data* dictionary, which maps variable names to list of their values; the *plot.addData* function is used to add them.

```
Yade [58]: from yade import plot

Yade [59]: plot.data
Out[59]: {}

Yade [60]: plot.addData(sigma=12,eps=1e-4)

# not adding sigma will add a NaN automatically
# this assures all variables have the same number of records
Yade [61]: plot.addData(eps=1e-3)

# adds NaNs to already existing sigma and eps columns
Yade [62]: plot.addData(force=1e3)

Yade [63]: plot.data
Out[63]:
{'sigma': [12, nan, nan],
 'eps': [0.0001, 0.001, nan],
 'force': [nan, nan, 1000.0]}

# retrieve only one column
Yade [64]: plot.data['eps']
Out[64]: [0.0001, 0.001, nan]

# get maximum eps
Yade [65]: max(plot.data['eps'])
Out[65]: 0.001
```

New record is added to all columns at every time *plot.addData* is called; this assures that lines in different columns always match. The special value `nan` or `NaN` (Not a Number) is inserted to mark the record invalid.

Note

It is not possible to have two columns with the same name, since data are stored as a dictionary.

To record data periodically, use *PyRunner*. This will record the z coordinate and velocity of body #1, iteration number and simulation time (every 20 iterations):

```
O.engines=O.engines+[PyRunner(command='myAddData()', iterPeriod=20)]

from yade import plot
def myAddData():
        b=O.bodies[1]
        plot.addData(z1=b.state.pos[2], v1=b.state.vel.norm(), i=O.iter, t=O.time)
```

Note

Arbitrary string can be used as a column label for *plot.data*. However if the name has spaces inside (e.g. `my funny column`) or is a reserved python keyword (e.g. `for`) the only way to pass it to *plot.addData* is to use a dictionary:

```
plot.addData(**{'my funny column':1e3, 'for':0.3})
```

An exception are columns having leading of trailing whitespaces. They are handled specially in *plot.plots* and should not be used (see below).

Labels can be conveniently used to access engines in the `myAddData` function:

```
O.engines=[...,
        UniaxialStrainer(...,label='strainer')
]
def myAddData():
        plot.addData(sigma=strainer.avgStress,eps=strainer.strain)
```

In that case, naturally, the labeled object must define attributes which are used (*UniaxialStrainer.strain* and *UniaxialStrainer.avgStress* in this case).

Plotting variables

Above, we explained how to track variables by storing them using *plot.addData*. These data can be readily used for plotting. Yade provides a simple, quick to use, plotting in the *plot* module. Naturally, since direct access to underlying data is possible via *plot.data*, these data can be processed in any other way.

The *plot.plots* dictionary is a simple specification of plots. Keys are x-axis variable, and values are tuple of y-axis variables, given as strings that were used for *plot.addData*; each entry in the dictionary represents a separate figure:

```
plot.plots={
        'i':('t',),     # plot t(i)
        't':('z1','v1') # z1(t) and v1(t)
}
```

Actual plot using data in *plot.data* and plot specification of *plot.plots* can be triggered by invoking the *plot.plot* function.

Live updates of plots

Yade features live-updates of figures during calculations. It is controlled by following settings:

- *plot.live* - By setting `yade.plot.live=True` you can watch the plot being updated while the calculations run. Set to `False` otherwise.
- *plot.liveInterval* - This is the interval in seconds between the plot updates.
- *plot.autozoom* - When set to `True` the plot will be automatically rezoomed.

Controlling line properties

In this subsection let us use a *basic complete script* like examples/simple-scene/simple-scene-plot.py, which we will later modify to make the plots prettier. Line of interest from that file is, and generates a picture presented below:

```
plot.plots={'i':('t'),'t':('z_sph',None,('v_sph','go-'),'z_sph_half')}
```

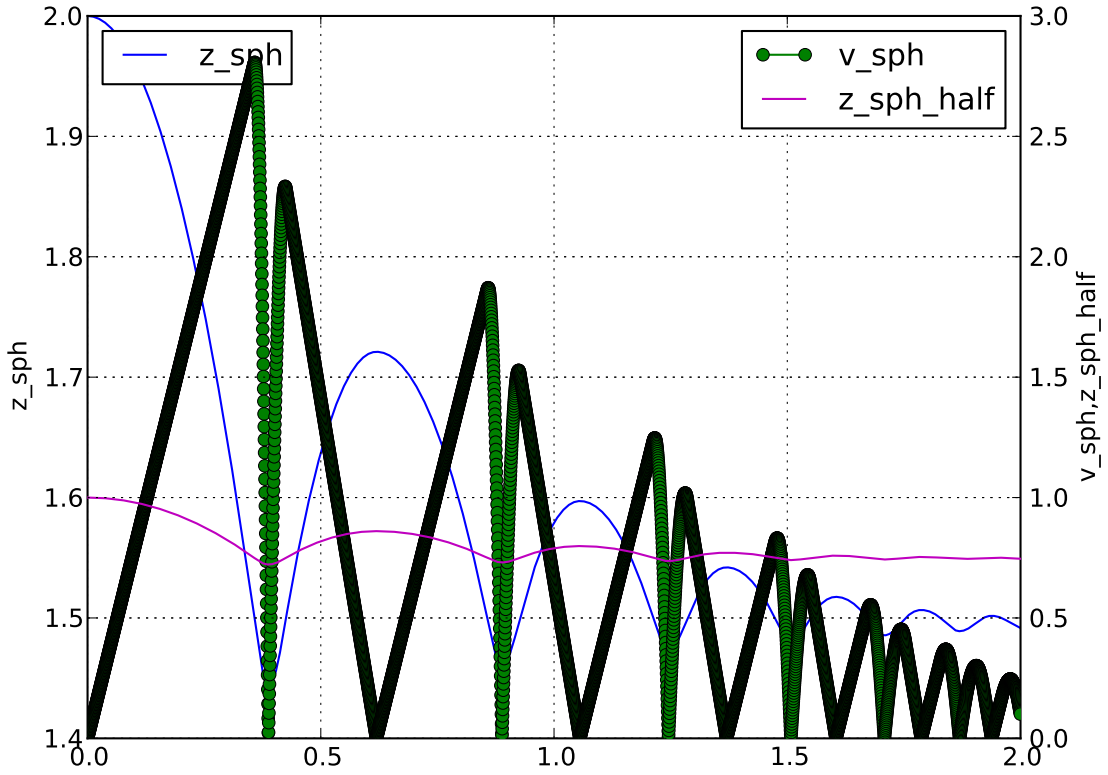


Fig. 12: Figure generated by examples/simple-scene/simple-scene-plot.py.

The line plots take an optional second string argument composed of a line color (eg. `'r'`, `'g'` or `'b'`), a line style (eg. `'-'`, `'--'` or `':'`) and a line marker (`'o'`, `'s'` or `'d'`). A red dotted line with circle markers is created with 'ro:' argument. For a listing of all options please have a look at http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib.pyplot.plot

For example using following plot.plots() command, will produce a following graph:

```
plot.plots={'i':(('t','xr:'),),'t':(('z_sph','r:'),None,('v_sph','g--'),('z_sph_half',
↪'b-.'))}
```

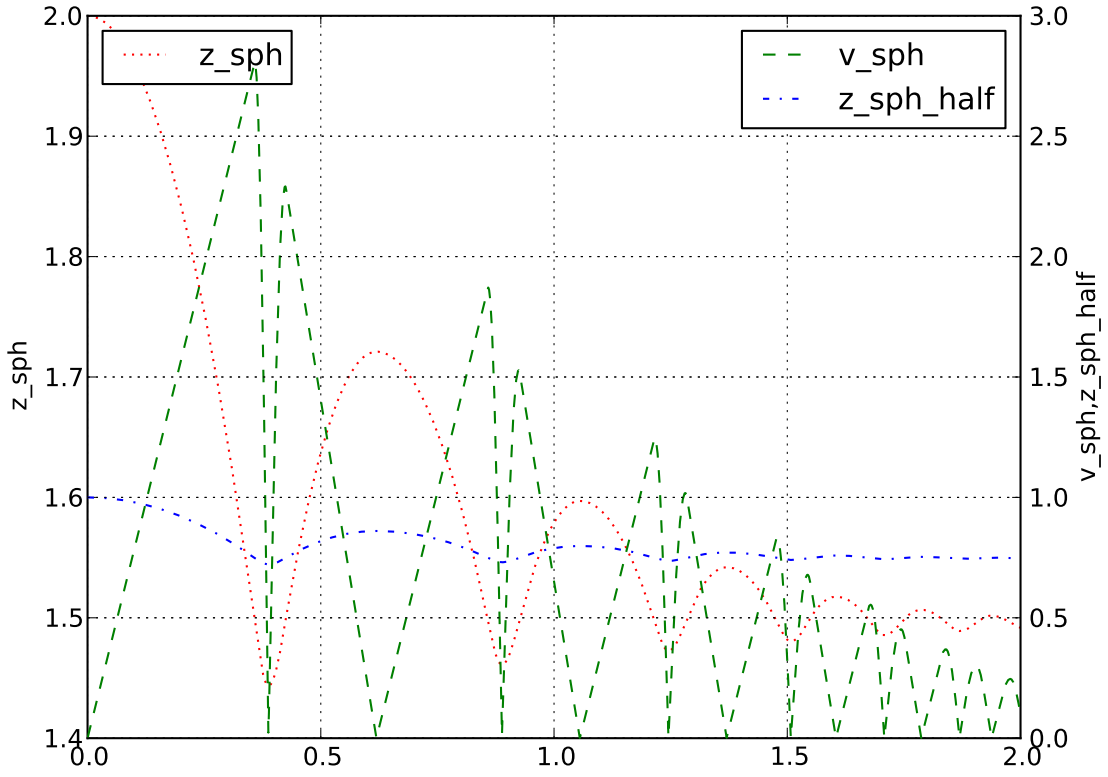


Fig. 13: Figure generated by changing parameters to plot.plots as above.

And this one will produce a following graph:

```
plot.plots={'i':(('t','xr:'),),'t':(('z_sph','Hr:'),None,('v_sph','+g--'),('z_sph_half
↪','*b-.'))}
```

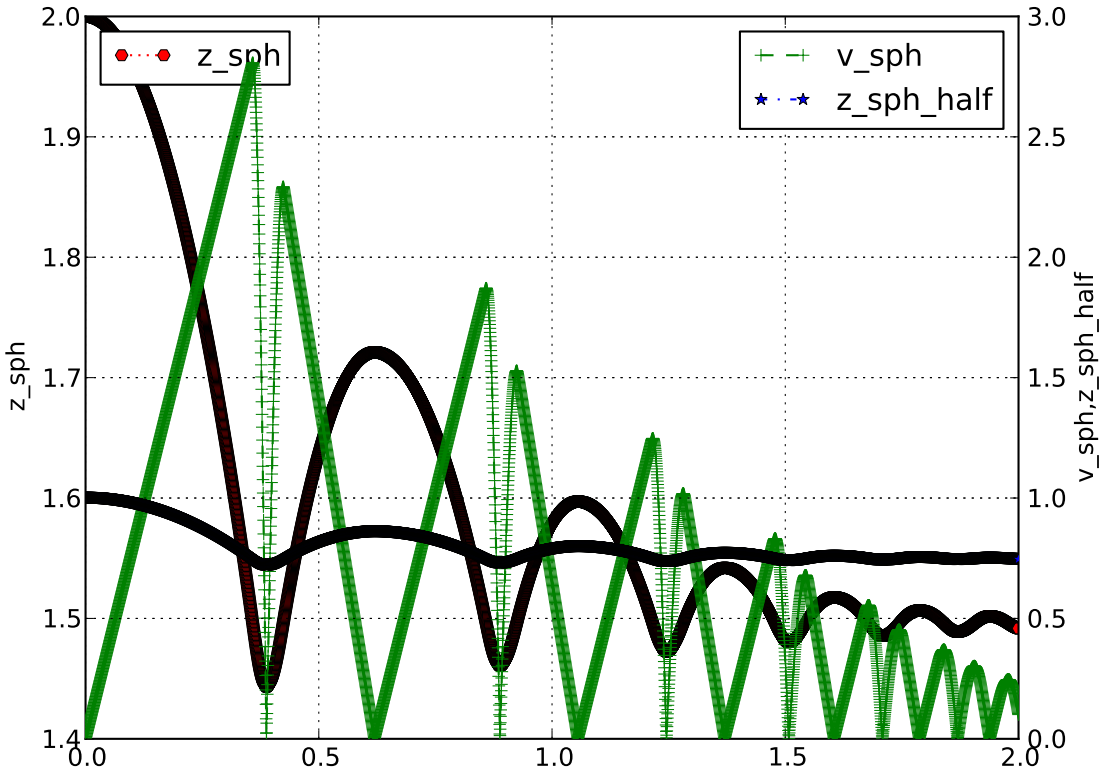


Fig. 14: Figure generated by changing parameters to plot.plots as above.

Note

You can learn more in matplotlib tutorial http://matplotlib.sourceforge.net/users/pyplot_tutorial.html and documentation http://matplotlib.sourceforge.net/users/pyplot_tutorial.html#controlling-line-properties

Note

Please note that there is an extra , in `'i':(('t','xr:'),)`, otherwise the `'xr:'` wouldn't be recognized as a line style parameter, but would be treated as an extra data to plot.

Controlling text labels

It is possible to use TeX syntax in plot labels. For example using following two lines in examples/simple-scene/simple-scene-plot.py, will produce a following picture:

```
plot.plots={'i':(('t','xr:'),),'t':(('z_sph','r:'),None,('v_sph','g--'),('z_sph_half',
↪'b-.'))}
```

(continues on next page)

(continued from previous page)

```
plot.labels={'z_sph':'$z_{sph}$' , 'v_sph':'$v_{sph}$' , 'z_sph_half':'$z_{sph}/2$'}
```

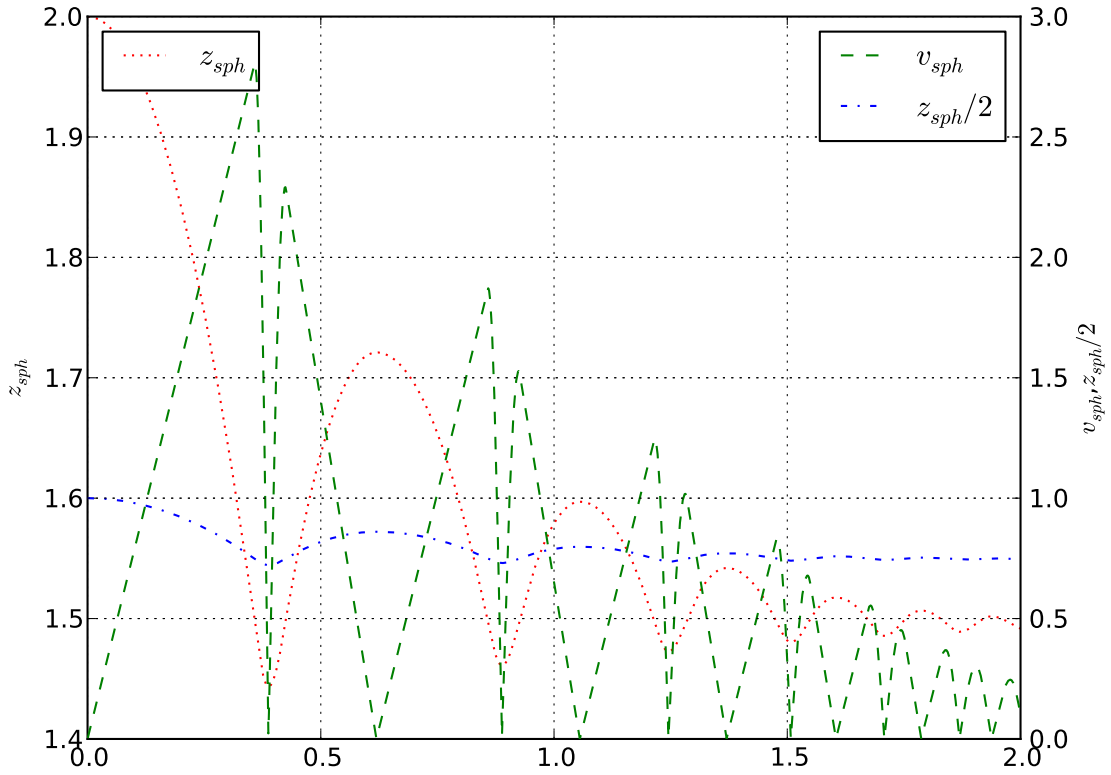


Fig. 15: Figure generated by examples/simple-scene/simple-scene-plot.py, with TeX labels.

Greek letters are simply a `'$\alpha$'`, `'$\beta$'` etc. in those labels. To change the font style a following command could be used:

```
yade.plot.matplotlib.rc('mathtext', fontset='stixsans')
```

But this is not part of yade, but a part of matplotlib, and if you want something more complex you really should have a look at matplotlib users manual http://matplotlib.sourceforge.net/users/index.html

Multiple figures

Since *plot.plots* is a dictionary, multiple entries with the same key (x-axis variable) would not be possible, since they overwrite each other:

```
Yade [66]: plot.plots={
   ....:    'i':('t',),
   ....:    'i':('z1','v1')
   ....: }
   ....:

Yade [67]: plot.plots
Out[67]: {'i': ('z1', 'v1')}
```

You can, however, distinguish them by prepending/appending space to the x-axis variable, which will be removed automatically when looking for the variable in *plot.data* – both x-axes will use the `i` column:

```
Yade [68]: plot.plots={
   ....:    'i':('t',),
   ....:    'i ':('z1','v1') # note the space in 'i '
   ....: }
   ....:

Yade [69]: plot.plots
Out[69]: {'i': ('t',), 'i ': ('z1', 'v1')}
```

Split y1 y2 axes

To avoid big range differences on the `y` axis, it is possible to have left and right `y` axes separate (like `axes x1y2` in gnuplot). This is achieved by inserting `None` to the plot specifier; variables coming before will be plot normally (on the left y-axis), while those after will appear on the right:

```
plot.plots={'i':('z1',None,'v1')}
```

Exporting

Plots and data can be exported to external files for later post-processing in Gnuplot via that *plot.saveGnuplot* function. Note that all data you added via plot.addData is saved - even data that you don't plot live during simulation. By editing the generated .gnuplot file you can plot any of the added Data afterwards.

- Data file is saved (compressed using bzip2) separately from the gnuplot file, so any other programs can be used to process them. In particular, the `numpy.genfromtxt` (documented here) can be useful to import those data back to python; the decompression happens automatically.
- The gnuplot file can be run through gnuplot to produce the figure; see *plot.saveGnuplot* documentation for details.

For post-processing with other tools than gnuplot, saved data can also be exported in another kind of text file with *plot.saveDataTxt*.

Stop conditions

For simulations with a pre-determined number of steps, it can be prescribed:

```
# absolute iteration number
O.stopAtIter=35466
O.run()
O.wait()
```

or

```
# number of iterations to run from now
O.run(35466,True) # wait=True
```

causes the simulation to run 35466 iterations, then stopping.

Frequently, decisions have to be made based on evolution of the simulation itself, which is not yet known. In such case, a function checking some specific condition is called periodically; if the condition is satisfied, `O.pause` or other functions can be called to stop the stimulation. See documentation for *Omega.run*, *Omega.pause*, *Omega.step*, *Omega.stopAtIter* for details.

For simulations that seek static equilibrium, the *unbalancedForce* can provide a useful metrics (see its documentation for details); for a desired value of `1e-2` or less, for instance, we can use:

```
def checkUnbalanced():
        if unbalancedForce<1e-2: O.pause()

O.engines=O.engines+[PyRunner(command="checkUnbalanced()",iterPeriod=100)]

# this would work as well, without the function defined apart:
#   PyRunner(command="if unablancedForce<1e-2: O.pause()",iterPeriod=100)

O.run(); O.wait()
# will continue after O.pause() will have been called
```

Arbitrary functions can be periodically checked, and they can also use history of variables tracked via *plot.addData*. For example, this is a simplified version of damage control in examples/concrete/uniax.py; it stops when current stress is lower than half of the peak stress:

```
O.engines=[...,
        UniaxialStrainer=(...,label='strainer'),
        PyRunner(command='myAddData()',iterPeriod=100),
        PyRunner(command='stopIfDamaged()',iterPeriod=100)
]

def myAddData():
        plot.addData(t=O.time,eps=strainer.strain,sigma=strainer.stress)

def stopIfDamaged():
        currSig=plot.data['sigma'][-1] # last sigma value
        maxSig=max(plot.data['sigma']) # maximum sigma value
        # print something in any case, so that we know what is happening
        print(plot.data['eps'][-1],currSig)
        if currSig<.5*maxSig:
                print("Damaged, stopping")
                print('gnuplot',plot.saveGnuplot(O.tags['id']))
                import sys
                sys.exit(0)

O.run(); O.wait()
# this place is never reached, since we call sys.exit(0) directly
```

Checkpoints

Occasionally, it is useful to revert to simulation at some past point and continue from it with different parameters. For instance, tension/compression test will use the same initial state but load it in 2 different directions. Two functions, *Omega.saveTmp* and *Omega.loadTmp* are provided for this purpose; *memory* is used as storage medium, which means that saving is faster, and also that the simulation will disappear when Yade finishes.

```
O.saveTmp()
# do something
O.saveTmp('foo')
O.loadTmp()      # loads the first state
O.loadTmp('foo') # loads the second state
```

Warning

`O.loadTmp` cannot be called from inside an engine, since *before* loading a simulation, the old one must finish the current iteration; it would lead to deadlock, since `O.loadTmp` would wait for the current

iteration to finish, while the current iteration would be blocked on `O.loadTmp`.

A special trick must be used: a separate function to be run after the current iteration is defined and is invoked from an independent thread launched only for that purpose:

```
O.engines=[...,PyRunner('myFunc()',iterPeriod=345)]

def myFunc():
        if someCondition:
                import thread
                # the () are arguments passed to the function
                thread.start_new_thread(afterIterFunc,())
def afterIterFunc():
        O.pause(); O.wait() # wait till the iteration really finishes
        O.loadTmp()

O.saveTmp()
O.run()
```

Remote control

Yade can be controlled remotely over network. At yade startup, the following lines appear, among other messages:

```
TCP python prompt on localhost:9000, auth cookie `dcekyu'
TCP info provider on localhost:21000
```

They inform about 2 ports on which connection of 2 different kind is accepted.

Python prompt

`TCP python prompt` is telnet server with authenticated connection, providing full python command-line. It listens on port 9000, or higher if already occupied (by another yade instance, for example).

Using the authentication cookie, connection can be made using telnet:

```
$ telnet localhost 9000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Enter auth cookie: dcekyu

__   __    ____                    __  _____ ____ ____
\ \ / /_ _|  _ \  ___    ___    / / |_   _/ ___|  _ \
 \ V / _` | | | |/ _ \  / _ \ / /    | || |   | |_) |
  | | (_| | |_| |  __/ | (_) / /     | || |___|  __/
  |_|\__,_|____/ \___|  \___/_/      |_| \____|_|

(connected from 127.0.0.1:40372)
>>>
```

The python pseudo-prompt `>>>` lets you write commands to manipulate simulation in variety of ways as usual. Two things to notice:

1. The new python interpreter (`>>>`) lives in a namespace separate from `Yade [1]:` command-line. For your convenience, `from yade import *` is run in the new python instance first, but local and global variables are not accessible (only builtins are).
2. The (fake) `>>>` interpreter does not have rich interactive feature of IPython, which handles the usual command-line `Yade [1]:`; therefore, you will have no command history, `?` help and so on.

Note

By giving access to python interpreter, full control of the system (including reading user's files) is possible. For this reason, **connection is only allowed from localhost**, not over network remotely. Of course you can log into the system via SSH over network to get remote access.

Warning

Authentication cookie is trivial to crack via bruteforce attack. Although the listener stalls for 5 seconds after every failed login attempt (and disconnects), the cookie could be guessed by trial-and-error during very long simulations on a shared computer.

Info provider

`TCP Info provider` listens at port 21000 (or higher) and returns some basic information about current simulation upon connection; the connection terminates immediately afterwards. The information is python dictionary represented as string (serialized) using standard pickle module.

This functionality is used by the batch system (described below) to be informed about individual simulation progress and estimated times. If you want to access this information yourself, you can study core/main/yade-batch.in for details.

MCP bridge

The MCP bridge allows AI agent clients to control Yade simulations via the Model Context Protocol, using a WebSocket connection that carries JSON messages. It exposes a small set of structured operations for executing code and managing long-running simulation tasks.

The bridge is distributed as a separate package; see the yade-mcp project for installation. Once the package is importable from Yade's embedded Python, the bridge is started from the Yade prompt:

```
Yade [1]: import yade_mcp_bridge
Yade [2]: yade_mcp_bridge.start()
YADE MCP Bridge on ws://localhost:9002, log: <cwd>/.yade-mcp/bridge.log
```

The bridge runs in the same Python namespace as the interactive console, so `O`, `O.bodies` and the rest of the simulation state are shared between the agent and the user. A separate `yade-mcp` MCP server process mediates between the agent client and this WebSocket.

Batch queuing and execution (yade-batch)

Yade features light-weight system for running one simulation with different parameters; it handles assignment of parameter values to python variables in simulation script, scheduling jobs based on number of available and required cores and more. The whole batch consists of 2 files:

simulation script
: regular Yade script, which calls *readParamsFromTable* to obtain parameters from parameter table. In order to make the script runnable outside the batch, *readParamsFromTable* takes default values of parameters, which might be overridden from the parameter table.

  readParamsFromTable knows which parameter file and which line to read by inspecting the `PARAM_TABLE` environment variable, set by the batch system.

parameter table
: simple text file, each line representing one parameter set. This file is read by *readParamsFromTable* (using *TableParamReader* class), called from simulation script, as explained above. For better reading of the text file you can make use of tabulators, these will be ignored by *readParamsFromTable*. Parameters are not restricted to numerical values. You can also make use of strings by `"quoting"` them (`' '` may also be used instead of `" "`). This can be useful for nominal parameters.

The batch can be run as

```
yade-batch parameters.table simulation.py
```

and it will intelligently run one simulation for each parameter table line. A minimal example is found in examples/test/batch/params.table and examples/test/batch/sim.py, another example follows.

Example

Suppose we want to study influence of parameters *density* and *initialVelocity* on position of a sphere falling on fixed box. We create parameter table like this:

```
description density initialVelocity # first non-empty line are column headings
reference   2400    10
hi_v           =    20              # = to use value from previous line
lo_v           =     5
# comments are allowed
hi_rho      5000    10
# blank lines as well:

hi_rho_v       =    20
hi_rh0_lo_v    =     5
```

Each line give one combination of these 2 parameters and assigns (which is optional) a *description* of this simulation.

In the simulation file, we read parameters from table, at the beginning of the script; each parameter has default value, which is used if not specified in the parameters file:

```
readParamsFromTable(
        gravity=-9.81,
        density=2400,
        initialVelocity=20,
        noTableOk=True     # use default values if not run in batch
)
from yade.params.table import *
print(gravity, density, initialVelocity)
```

after the call to *readParamsFromTable*, corresponding python variables are created in the `yade.params.table` module and can be readily used in the script, e.g.

```
GravityEngine(gravity=(0,0,gravity))
```

Let us see what happens when running the batch:

```
$ yade-batch batch.table batch.py
Will run `/usr/local/bin/yade-trunk' on `batch.py' with nice value 10, output␣
↪redirected to `batch.@.log', 4 jobs at a time.
Will use table `batch.table', with available lines 2, 3, 4, 5, 6, 7.
Will use lines  2 (reference), 3 (hi_v), 4 (lo_v), 5 (hi_rho), 6 (hi_rho_v), 7␣
↪(hi_rh0_lo_v).
Master process pid 7030
```

These lines inform us about general batch information: nice level, log file names, how many cores will be used (4); table name, and line numbers that contain parameters; finally, which lines will be used; master PID is useful for killing (stopping) the whole batch with the `kill` command.

```
Job summary:
   #0 (reference/4): PARAM_TABLE=batch.table:2 DISPLAY=  /usr/local/bin/yade-trunk --
```

(continues on next page)

(continued from previous page)

```
↪threads=4 --nice=10 -x batch.py > batch.reference.log 2>&1
  #1 (hi_v/4): PARAM_TABLE=batch.table:3 DISPLAY=  /usr/local/bin/yade-trunk --
↪threads=4 --nice=10 -x batch.py > batch.hi_v.log 2>&1
  #2 (lo_v/4): PARAM_TABLE=batch.table:4 DISPLAY=  /usr/local/bin/yade-trunk --
↪threads=4 --nice=10 -x batch.py > batch.lo_v.log 2>&1
  #3 (hi_rho/4): PARAM_TABLE=batch.table:5 DISPLAY=  /usr/local/bin/yade-trunk --
↪threads=4 --nice=10 -x batch.py > batch.hi_rho.log 2>&1
  #4 (hi_rho_v/4): PARAM_TABLE=batch.table:6 DISPLAY=  /usr/local/bin/yade-trunk --
↪threads=4 --nice=10 -x batch.py > batch.hi_rho_v.log 2>&1
  #5 (hi_rh0_lo_v/4): PARAM_TABLE=batch.table:7 DISPLAY=  /usr/local/bin/yade-trunk -
↪-threads=4 --nice=10 -x batch.py > batch.hi_rh0_lo_v.log 2>&1
```

displays all jobs with command-lines that will be run for each of them. At this moment, the batch starts to be run.

```
#0 (reference/4) started on Tue Apr 13 13:59:32 2010
#0 (reference/4) done    (exit status 0), duration 00:00:01, log batch.reference.log
#1 (hi_v/4) started on Tue Apr 13 13:59:34 2010
#1 (hi_v/4) done    (exit status 0), duration 00:00:01, log batch.hi_v.log
#2 (lo_v/4) started on Tue Apr 13 13:59:35 2010
#2 (lo_v/4) done    (exit status 0), duration 00:00:01, log batch.lo_v.log
#3 (hi_rho/4) started on Tue Apr 13 13:59:37 2010
#3 (hi_rho/4) done    (exit status 0), duration 00:00:01, log batch.hi_rho.log
#4 (hi_rho_v/4) started on Tue Apr 13 13:59:38 2010
#4 (hi_rho_v/4) done    (exit status 0), duration 00:00:01, log batch.hi_rho_v.log
#5 (hi_rh0_lo_v/4) started on Tue Apr 13 13:59:40 2010
#5 (hi_rh0_lo_v/4) done    (exit status 0), duration 00:00:01, log␣
↪batch.hi_rh0_lo_v.log
```

information about job status changes is being printed, until:

```
All jobs finished, total time  00:00:08
Log files:
batch.reference.log batch.hi_v.log batch.lo_v.log batch.hi_rho.log batch.hi_rho_v.log␣
↪batch.hi_rh0_lo_v.log
Bye.
```

Separating output files from jobs

As one might output data to external files during simulation (using classes such as *VTKRecorder*), it is important to name files in such way that they are not overwritten by next (or concurrent) job in the same batch. A special tag `O.tags['id']` is provided for such purposes: it is comprised of date, time and PID, which makes it always unique (e.g. `20100413T144723p7625`); additional advantage is that alphabetical order of the `id` tag is also chronological. To add the used parameter set or the description of the job, if set, you could add O.tags[‘params’] to the filename.

For smaller simulations, prepending all output file names with `O.tags['id']` can be sufficient:

```
saveGnuplot(O.tags['id'])
```

For larger simulations, it is advisable to create separate directory of that name first, putting all files inside afterwards:

```
os.mkdir(O.tags['id'])
O.engines=[
        # …
        VTKRecorder(fileName=O.tags['id']+'/'+'vtk'),
```

(continues on next page)

(continued from previous page)

```
        # …
]
# …
O.saveGnuplot(O.tags['id']+'/'+'graph1')
```

Controlling parallel computation

Default total number of available cores is determined from `/proc/cpuinfo` (provided by Linux kernel); in addition, if `OMP_NUM_THREADS` environment variable is set, minimum of these two is taken. The `-j`/`--jobs` option can be used to override this number.

By default, each job uses all available cores for itself, which causes jobs to be effectively run in parallel. Number of cores per job can be globally changed via the `--job-threads` option.

Table column named `!OMP_NUM_THREADS` (`!` prepended to column generally means to assign *environment variable*, rather than python variable) controls number of threads for each job separately, if it exists.

If number of cores for a job exceeds total number of cores, warning is issued and only the total number of cores is used instead.

Merging gnuplot from individual jobs

Frequently, it is desirable to obtain single figure for all jobs in the batch, for comparison purposes. Somewhat heuristic way for this functionality is provided by the batch system. `yade-batch` must be run with the `--gnuplot` option, specifying some file name that will be used for the merged figure:

```
yade-trunk --gnuplot merged.gnuplot batch.table batch.py
```

Data are collected in usual way during the simulation (using *plot.addData*) and saved to gnuplot file via *plot.saveGnuplot* (it creates 2 files: gnuplot command file and compressed data file). The batch system *scans*, once the job is finished, log file for line of the form `gnuplot [something]`. Therefore, in order to print this *magic line* we put:

```
print('gnuplot',plot.saveGnuplot(O.tags['id']))
```

and the end of the script (even after waitIfBatch()) , which prints:

```
gnuplot 20100413T144723p7625.gnuplot
```

to the output (redirected to log file).

This file itself contains single graph:

At the end, the batch system knows about all gnuplot files and tries to merge them together, by assembling the `merged.gnuplot` file.

HTTP overview

While job is running, the batch system presents progress via simple HTTP server running at port 9080, which can be acessed from a regular web browser (or e.g. `lynx` for a terminal usage) by requesting the `http://localhost:9080` URL. This page can be accessed remotely over network as well.

Batch execution on Job-based clusters (OAR)

On High Performance Computation (HPC) clusters with a scheduling system, the following script might be useful. Exactly like yade-batch, it handles assignemnt of parameters value to python variables in simulation script from a parameter table, and job submission. This script is written for oar-based system , and may be extended to others ones. On those system, usually, a job can't run forever and has a specific duration allocation. The whole job submission consists of 3 files:

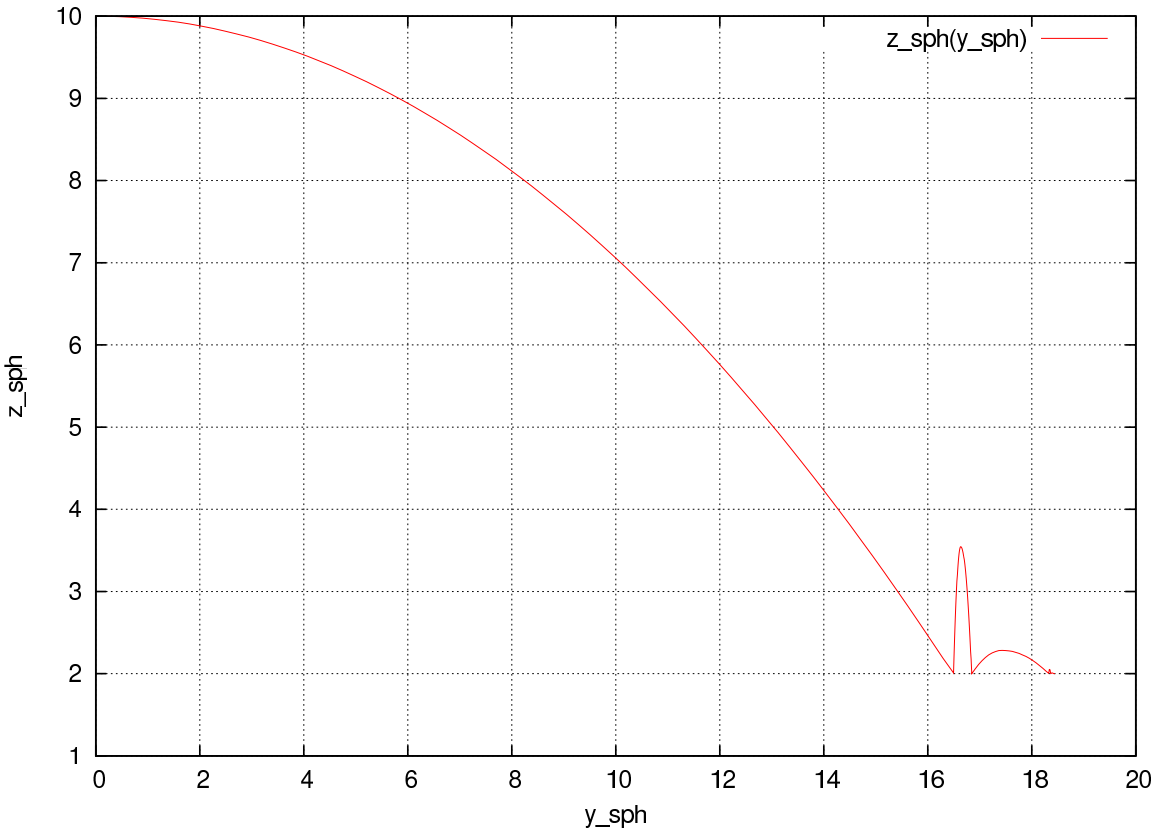


Fig. 16: Figure from single job in the batch.

Simulation script:
Regular Yade script, which calls *readParamsFromTable* to obtain parameters from parameter table. In order to make the script runnable outside the batch, *readParamsFromTable* takes default values of parameters, which might be overridden from the parameter table.

readParamsFromTable knows which parameter file and which line to read by inspecting the `PARAM_TABLE` environment variable, set by the batch system.

Parameter table:
Simple text file, each line representing one parameter set. This file is read by *readParamsFromTable* (using *TableParamReader* class), called from simulation script, as explained above. For better reading of the text file you can make use of tabulators, these will be ignored by *readParamsFromTable*. Parameters are not restricted to numerical values. You can also make use of strings by `"quoting"` them (`' '` may also be used instead of `" "`). This can be useful for nominal parameters.

Job script:
Bash script, which calls yade on computing nodes. This script eventually creates temp folders, save data to storage server etc. The script must be formatted as a template where some tags will be replaced by specific values at the execution time:

- `__YADE_COMMAND__` will be replaced by the actual yade run command
- `__YADE_LOGFILE__` will be replaced by the log file path (output to stdout)
- `__YADE_ERRFILE__` will be replaced by the error file path (output to stderr)
- `__YADE_JOBNO__` will be replaced by an identifier composed as (launch script pid)-(job order)
- `__YADE_JOBID__` will be replaced by an identifier composed of all parameters values

The batch can be run as

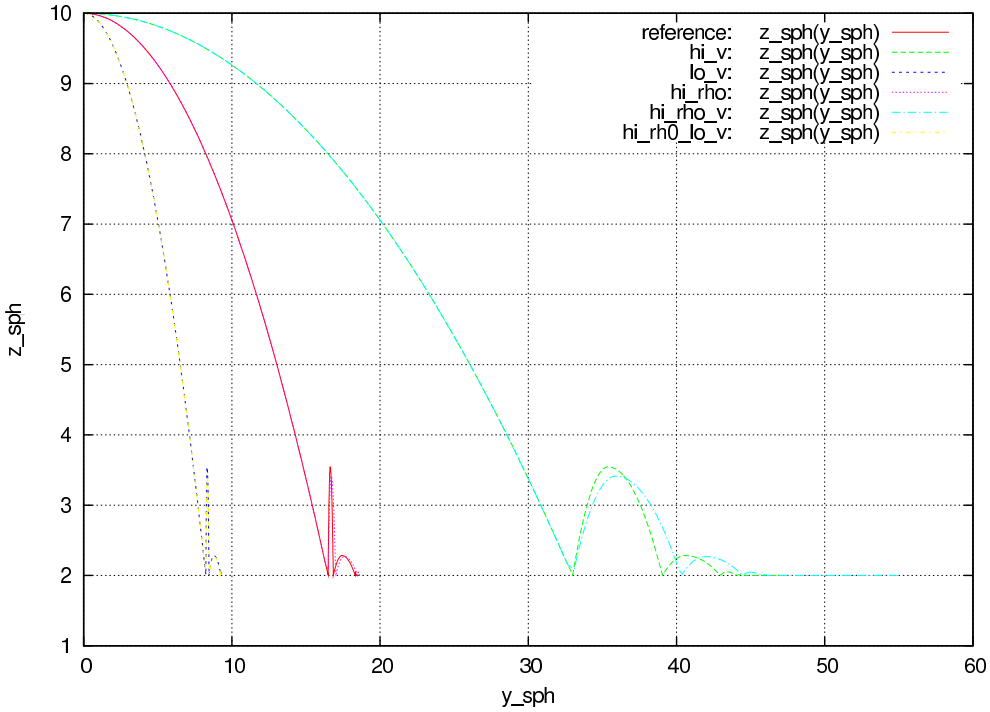


Fig. 17: Merged figure from all jobs in the batch. Note that labels are prepended by job description to make lines distinguishable.

Running for 00:10:19, since Tue Apr 13 16:17:11 2010.

Pid 9873

4 slots available, 4 used, 0 free.

Jobs

4 total, **2** running, **1** done

id	status	info	slots	command
_geomType=B	00:10:19	96.33% done step 9180/9530 avg 14.9596/sec 10267 bodies 65506 intrs	2	PARAM_TABLE=iParams.table:2 DISPLAY= /usr/local/bin/yade-trunk --threads=2 --nice=10 -x indent.py > indent._geomType=B.log 2>&1
_geomType=smallA	00:09:53	(no info)	2	PARAM_TABLE=iParams.table:3 DISPLAY= /usr/local/bin/yade-trunk --threads=2 --nice=10 -x indent.py > indent._geomType=smallA.log 2>&1
_geomType=smallB	00:00:24	6.95% done step 694/9985 avg 35.8212/sec 9021 bodies 58352 intrs	2	PARAM_TABLE=iParams.table:4 DISPLAY= /usr/local/bin/yade-trunk --threads=2 --nice=10 -x indent.py > indent._geomType=smallB.log 2>&1
_geomType=smallC	(pending)	(no info)	2	PARAM_TABLE=iParams.table:5 DISPLAY= /usr/local/bin/yade-trunk --threads=2 --nice=10 -x indent.py > indent._geomType=smallC.log 2>&1

Fig. 18: Summary page available at port 9080 as batch is processed (updates every 5 seconds automatically). Possible job statuses are pending, running, done, failed.

```
yade-oar --oar-project=<your project name> --oar-script=job.sh --oar-
↪walltime=hh:mm:ss parameters.table simulation.py
```

and it will generate one launch script and submit one job for each parameter table line. A minimal example is found in examples/oar/params.table examples/oar/job.sh and examples/oar/sim.py.

Note

You have to specify either –oar-walltime or a !WALLTIME column in params.table. !WALLTIME will override –oar-walltime

Warning

yade-oar is not compiled by default, use -DENABLE_OAR=1 option to cmake to enable it. Please note also that submitting yade jobs (or yade-batch jobs) through OAR does not actually require to use yade-oar. The point of yade-oar is about making yade submit a batch of OAR jobs, instead of sumbitting a yade batch as one OAR job. Mind that it may be viewed as a hack of the OAR scheduler itself by some HPC admins.

2.2.3 Postprocessing

3d rendering & videos

3D rendering is available at runtime to inspect the simulation, it uses QGLViewer library.

Many parameters of the rendering can be modified. See for instance background color *bgColor* in *OpenGLRenderer* - also visible in the "Display" tab of Qt Controller, and the keys listed in QGLViewer's help (hit "h" when the 3D window is active). For instance, hit "t" to switch between orthographic / perspective camera, or "m" to move particle around with the mouse.

colorStyle helps to quickly switch between predefined styles for particle color, background color, and resolution of sphere representation. For images to be included in documents it is generally better to use a white background:

```
colorStyle.setStyle("figureColor",True)  # colorful particles on white background
colorStyle.setStyle("figureGrey",True)  # grey levels for everything
```

These two styles also have higher resolution for displaying the spheres. The style that was used pre-2024 is *old*. The available styles are listed in *colorStyle.styles*:

```
Yade [70]: colorStyle.styles
Out[70]:
{'sand': <yade.utils.YadeColorStyle at 0x7f980889cc20>,
 'old': <yade.utils.YadeColorStyle at 0x7f98087f3b10>,
 'figureColor': <yade.utils.YadeColorStyle at 0x7f98087f3c50>,
 'figureGrey': <yade.utils.YadeColorStyle at 0x7f980885d220>,
 'blue': <yade.utils.YadeColorStyle at 0x7f980885d350>,
 'screenDisplayLowRes': <yade.utils.YadeColorStyle at 0x7f98089a7d10>}
```

3D rendering is one of several ways to produce videos of simulations:

1. Capture screen output (the 3d rendering window) during the simulation — there are tools available for that (such as Istanbul or RecordMyDesktop, which are also packaged for most Linux distributions). The output is "what you see is what you get", with all the advantages and disadvantages.
2. Periodic frame snapshot using *SnapshotEngine* (see examples/test/force-network-video.py, examples/bulldozer/bulldozer.py or examples/test/beam-l6geom.py for a complete example):

```
O.engines=[
  #...
  SnapshotEngine(iterPeriod=100,fileBase='/tmp/bulldozer-',viewNo=0,label=
↪'snapshooter')
]
```

which will save numbered files like `/tmp/bulldozer-0000.png`. These files can be processed externally (with mencoder and similar tools) or directly with the *makeVideo*:

```
makeVideo(frameSpec,out,renameNotOverwrite=True,fps=24,kbps=6000,bps=None)
```

The video is encoded using the default mencoder codec (mpeg4).

3. Specialized post-processing tools, notably Paraview. This is described in more detail in the following section.

Paraview

Saving data during the simulation

Paraview is based on the Visualization Toolkit, which defines formats for saving various types of data. One of them (with the `.vtu` extension) can be written by a special engine *VTKRecorder*. It is added to the simulation loop:

```
O.engines=[
        # ...
        VTKRecorder(iterPeriod=100,recorders=['spheres','facets','colors'],fileName='/
↪tmp/p1-')
]
```

- *iterPeriod* determines how often to save simulation data (besides *iterPeriod*, you can also use *virtPeriod* or *realPeriod*). If the period is too high (and data are saved only few times), the video will have few frames.
- *fileName* is the prefix for files being saved. In this case, output files will be named `/tmp/p1-spheres.0.vtu` and `/tmp/p1-facets.0.vtu`, where the number is the number of iteration; many files are created, putting them in a separate directory is advisable.
- *recorders* determines what data to save

export.VTKExporter plays a similar role, with the difference that it is more flexible. It will save any user defined variable associated to the bodies.

Loading data into Paraview

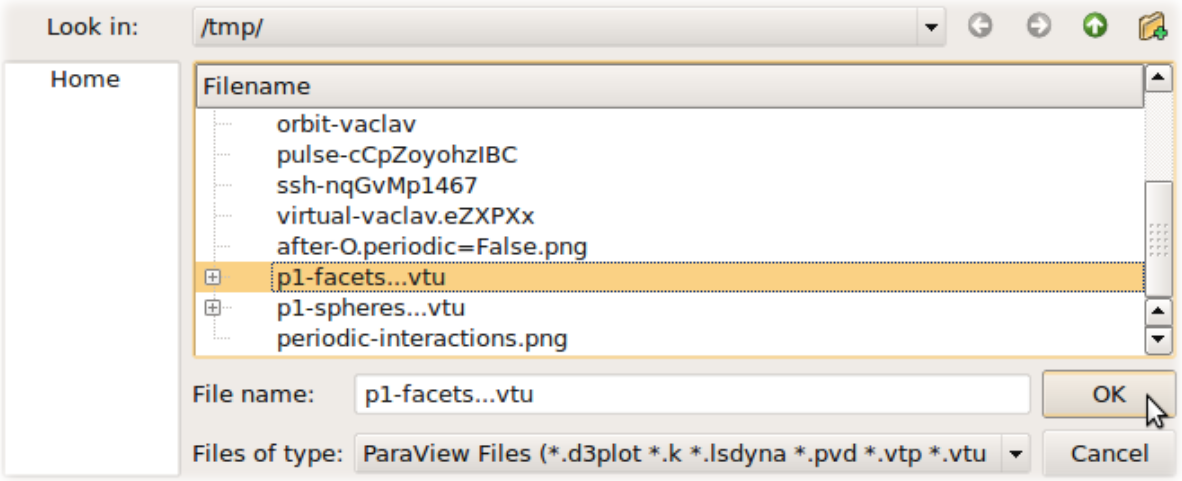
All sets of files (`spheres`, `facets`, ...) must be opened one-by-one in Paraview. The open dialogue automatically collapses numbered files in one, making it easy to select all of them:

Click on the "Apply" button in the "Object inspector" sub-window to make loaded objects visible. You can see tree of displayed objects in the "Pipeline browser":

Rendering spherical particles. Glyphs

Spheres will only appear as points. To make them look as spheres, you have to add "glyph" to the `p1-spheres.*` item in the pipeline using the icon. Then set (in the Object inspector)

- "Glyph type" to *Sphere*
- "Radius" to *1*



- "Scale mode" to *Scalar* (*Scalar* is set above to be the *radii* value saved in the file, therefore spheres with radius *1* will be scaled by their true radius)
- "Set scale factor" to *1*
- optionally uncheck "Mask points" and "Random mode" (they make some particles not to be rendered for performance reasons, controlled by the "Maximum Number of Points")

After clicking "Apply", spheres will appear. They will be rendered over the original white points, which you can disable by clicking on the eye icon next to `p1-spheres.*` in the Pipeline browser.

Rendering spherical particles. PointSprite

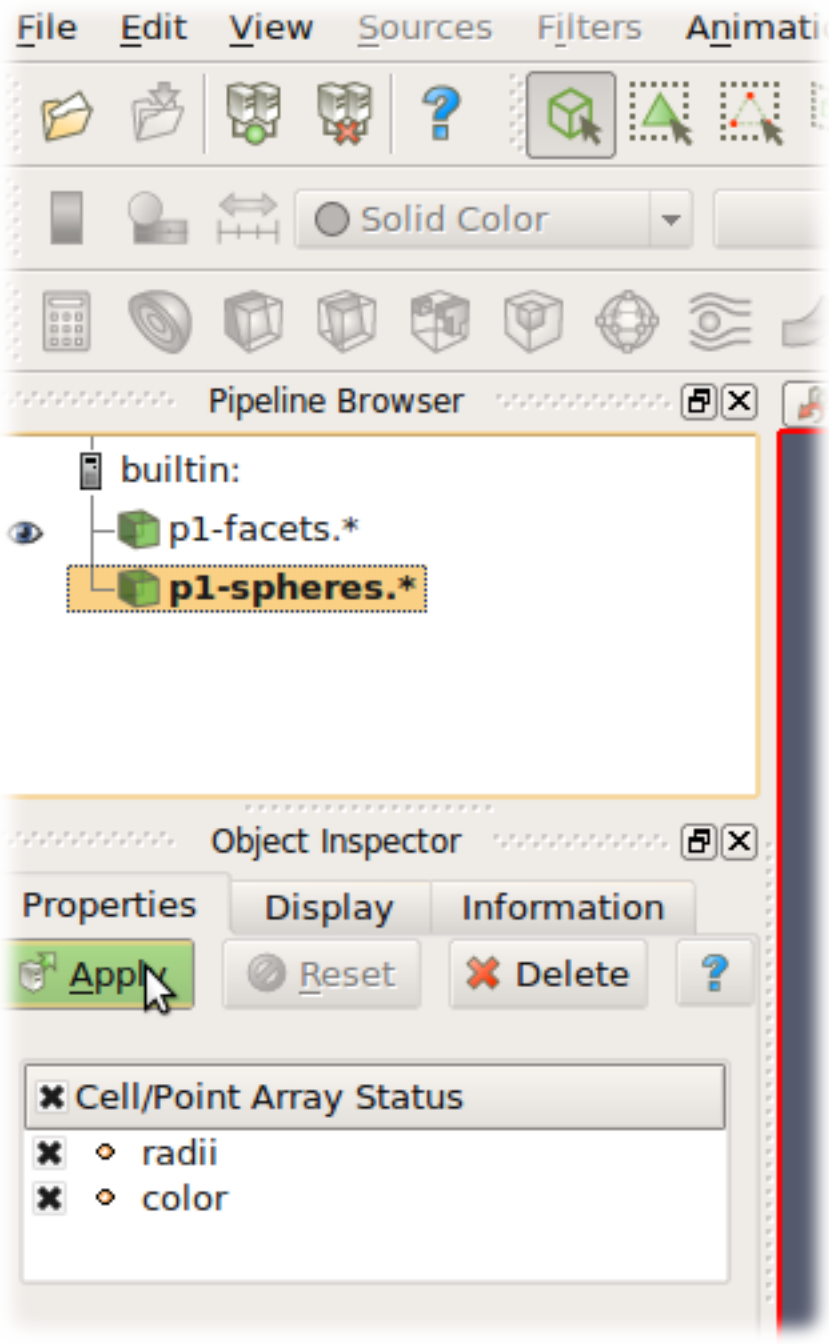
Another opportunity to display spheres is by using *PointSprite* plugin. This technique requires much less RAM in comparison to Glyphs.

- "Tools -> Manage Plugins"
- "PointSprite_Plugin -> Load selected -> Close"
- Load VTU-files
- "Representation -> Point Sprite"
- "Point Sprite -> Scale By -> radii"
- "Edit Radius Transfer Function -> Proportional -> Multiplier = 1.0 -> Close"

Rendering interactions as force chain

Data saved by `VTKRecorder` (the steps below generates cones rather than tubes) or `export.VTKExporter(...).exportInteractions(what=dict(forceN='i.phys.normalForce.norm()'))` (the steps below generates per interaction tubes with constant radius):

- Load interactions VTP or VTK files
- Filters -> Cell Data To Point Data
- Filters -> Tube
- Set color by "forceN"
- Set "Vary Radius" to "By Scalar"
- Set "Radius" and "Radius Factor" such that the result looks OK (in 3D postprocessing tutorial script, Radius=0.0005 and Radius Factor=100 looks reasonably)


File
Edit
View
Sources
Filters
Animati
Solid Color
Pipeline Browser
builtin:
p1-facets.*
p1-spheres.*
Object Inspector
Properties
Display
Information
Apply
Reset
Delete
Cell/Point Array Status
radii
color

Facet transparency

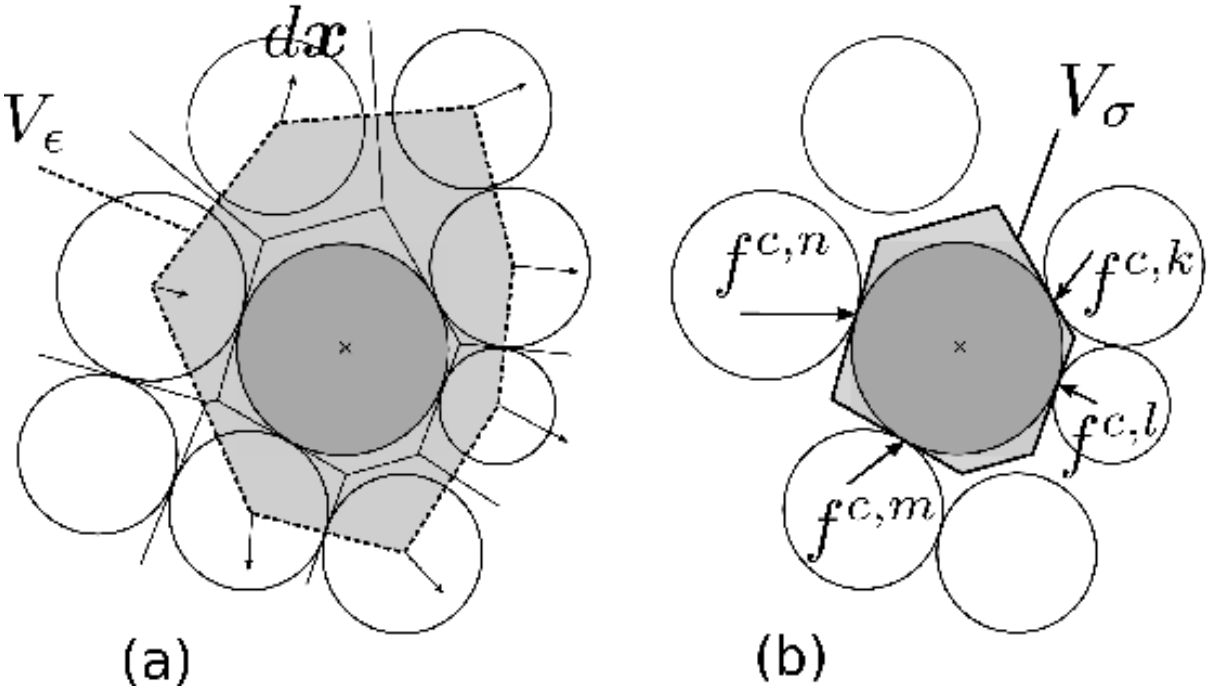
If you want to make facet objects transparent, select `p1-facets.*` in the Pipeline browser, then go to the Object inspector on the Display tab. Under "Style", you can set the "Opacity" value to something smaller than 1.

Animation

You can move between frames (snapshots that were saved) via the "Animation" menu. After setting the view angle, zoom etc to your satisfaction, the animation can be saved with *File/Save animation*.

Micro-stress and micro-strain

It is sometimes useful to visualize a DEM simulation through equivalent strain fields or stress fields. This is possible with *TesselationWrapper*. This class handles the triangulation of spheres in a scene, build tesselation on request, and give access to computed quantities: volume, porosity and local deformation for each sphere. The definition of microstrain and microstress is at the scale of particle-centered subdomains shown below, as explained in [Catalano2014a] .



Micro-strain

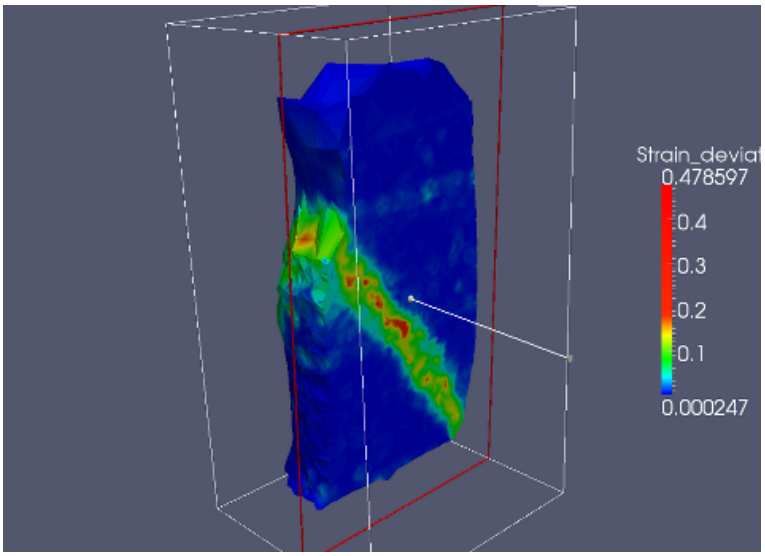
Below is an output of the *defToVtk* function visualized with paraview (in this case Yade's TesselationWrapper was used to process experimental data obtained on sand by Edward Ando at Grenoble University, 3SR lab.). The output is visualized with paraview, as explained in the previous section. Similar results can be generated from simulations:

```
tt=TriaxialTest()
tt.generate("test.yade")
O.load("test.yade")
O.run(100,True)
TW=TesselationWrapper()
TW.triangulate()        #compute regular Delaunay triangulation, don't construct␣
↪tesselation
TW.computeVolumes()     #will silently tesselate the packing, then compute volume of␣
↪each Voronoi cell
```

(continues on next page)

(continued from previous page)

```
TW.volume(10)           #get volume associated to sphere of id 10
TW.setState(0)          #store current positions internaly for later use as the "0"␣
↪state
O.run(100,True)         #make particles move a little (let's hope they will!)
TW.setState(1)          #store current positions internaly in the "1" (deformed) state
#Now we can define strain by comparing states 0 and 1, and average them at the␣
↪particles scale
TW.defToVtk("strain.vtk")
```



Micro-stress

Stress fields can be generated by combining the volume returned by TesselationWrapper to per-particle stress given by *bodyStressTensors*. Since the stress σ from bodyStressTensor implies a division by the volume V_b of the solid particle, one has to re-normalize it in order to obtain the micro-stress as defined in [Catalano2014a] (equation 39 therein), i.e. $\overline{\sigma}^k = \sigma^k \times V_b^k / V_\sigma^k$ where V_σ^k is the volume assigned to particle k in the tesselation. For instance:

```
#"b" being a body
TW=TesselationWrapper()
TW.setState()
TW.computeVolumes()
s=bodyStressTensors()
stress = s[b.id]*4.*pi/3.*b.shape.radius**3/TW.volume(b.id)
```

As any other value, the stress can be exported to a vtk file for display in Paraview using *export.VTKExporter*.

2.2.4 Python specialties and tricks

Importing Yade in other Python applications

Yade can be imported in other Python applications. To do so, you need somehow to make yade executable .py extended. The easiest way is to create a symbolic link, i.e. (suppose your Yade executable file is called "yade-trunk" and you want make it "yadeimport.py"):

```
$ cd /path/where/you/want/yadeimport
$ ln -s /path/to/yade/executable/yade-trunk yadeimport.py
```

Then you need to make your yadeimport.py findable by Python. You can export PYTHONPATH environment variable, or simply use sys.path directly in Python script:

```
import sys
sys.path.append('/path/where/you/want/yadeimport')
from yadeimport import *

print(Matrix3(1,2,3, 4,5,6, 7,8,9))
print(O.bodies)
# any other Yade code
```

2.2.5 Extending Yade

- new particle shape
- new constitutive law

2.2.6 Troubleshooting

Crashes

It is possible that you encounter crash of Yade, i.e. Yade terminates with error message such as

```
Segmentation fault (core dumped)
```

without further explanation. Frequent causes of such conditions are

- program error in Yade itself;
- fatal condition in your particular simulation (such as impossible dispatch);
- problem with graphics card driver.

Try to reproduce the error (run the same script) with debug-enabled version of Yade. Debugger will be automatically launched at crash, showing backtrace of the code (in this case, we triggered crash by hand):

```
Yade [1]: import os,signal
Yade [2]: os.kill(os.getpid(),signal.SIGSEGV)
SIGSEGV/SIGABRT handler called; gdb batch file is `/tmp/yade-YwtfRY/tmp-0'
GNU gdb (GDB) 7.1-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
[Thread debugging using libthread_db enabled]
[New Thread 0x7f0fb1268710 (LWP 16471)]
[New Thread 0x7f0fb29f2710 (LWP 16470)]
[New Thread 0x7f0fb31f3710 (LWP 16469)]

...
```

What looks as cryptic message is valuable information for developers to locate source of the bug. In particular, there is (usually) line `<signal handler called>`; lines below it are source of the bug (at least very likely so):

```
Thread 1 (Thread 0x7f0fcee53700 (LWP 16465)):
#0  0x00007f0fcd8f4f7d in __libc_waitpid (pid=16497, stat_loc=<value optimized out>,␣
↪options=0) at ../sysdeps/unix/sysv/linux/waitpid.c:41
#1  0x00007f0fcd88c7e9 in do_system (line=<value optimized out>) at ../
↪sysdeps/posix/system.c:149
#2  0x00007f0fcd88cb20 in __libc_system (line=<value optimized out>) at ../
↪sysdeps/posix/system.c:190
#3  0x00007f0fcd0b4b23 in crashHandler (sig=11) at core/main/pyboot.cpp:45
#4  <signal handler called>
#5  0x00007f0fcd87ed57 in kill () at ../sysdeps/unix/syscall-template.S:82
#6  0x000000000051336d in posix_kill (self=<value optimized out>, args=<value␣
↪optimized out>) at ../Modules/posixmodule.c:4046
#7  0x00000000004a7c5e in call_function (f=Frame 0x1c54620, for file <ipython␣
↪console>, line 1, in <module> (), throwflag=<value optimized out>) at ../
↪Python/ceval.c:3750
#8  PyEval_EvalFrameEx (f=Frame 0x1c54620, for file <ipython console>, line 1, in
↪<module> (), throwflag=<value optimized out>) at ../Python/ceval.c:2412
```

If you think this might be error in Yade, file a bug report as explained below. Do not forget to attach *full* yade output from terminal, including startup messages and debugger output – select with right mouse button, with middle button paste the bugreport to a file and attach it. Attach your simulation script as well.

Reporting bugs

Bugs are general name for defects (functionality shortcomings, misdocumentation, crashes) or feature requests. They are tracked at https://gitlab.com/yade-dev/trunk/issues.

When reporting a new bug, be as specific as possible; state version of yade you use, system version and the output of *printAllVersions()*, as explained in the above section on crashes.

2.2.7 Getting in touch with Yade community

Public questions and answers for getting help

Hint

Please use the GitLab interface for asking questions about Yade.

In case you're not familiar with computer oriented discussion lists, please read this wiki page (a Yade-oriented and shortened version of How To Ask Questions The Smart Way) before posting, in order to increase your chances getting help. Do not forget to state what *version* of Yade you use (shown when you start Yade, or even better as printed by function *libVersions.printAllVersions*), whether you installed it from source code or a package, what operating system (such as Ubuntu 18.04), and if you have done any local modifications to source code in case of compiled version.

Private and/or paid support

You might contact developers by their private email (rather than by the Launchpad interface or the mailing lists) or the generic adress consult|A|yade-dem.org for a closer, private, support. This is also a suitable method for proposing financial reward for implementation of a substantial feature that is not yet in Yade – typically, though, we will request this feature to be part of the public codebase once completed, so that the rest of the community can benefit from it as well.

Wiki

http://www.yade-dem.org/wiki/

Discord chat

https://discord.gg/rku35YXZJd

Twitter account

https://twitter.com/YadeDEM

2.3 Yade wrapper class reference

2.3.1 Bodies

Body

Shape

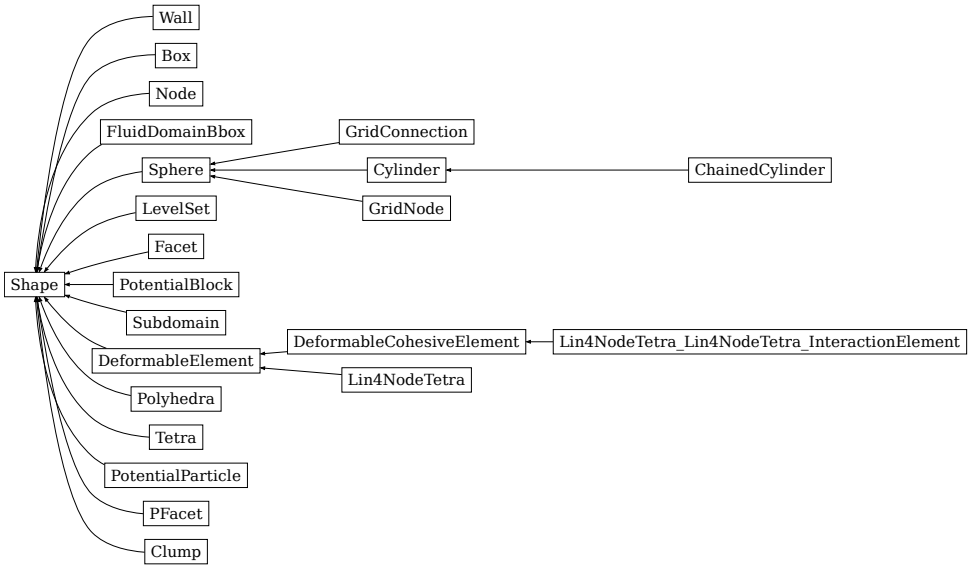


Fig. 19: Inheritance graph of Shape. See also: *Box*, *ChainedCylinder*, *Clump*, *Cylinder*, *DeformableCohesiveElement*, *DeformableElement*, *Facet*, *FluidDomainBbox*, *GridConnection*, *GridNode*, *LevelSet*, *Lin4NodeTetra*, *Lin4NodeTetra_Lin4NodeTetra_InteractionElement*, *Node*, *PFacet*, *Polyhedra*, *PotentialBlock*, *PotentialParticle*, *Sphere*, *Subdomain*, *Tetra*, *Wall*.

`class yade.wrapper.Box(`*inherits Shape → Serializable*`)`

`dict(`*(Serializable)arg1*`)` → dict :

Return dictionary of attributes.

`dispHierarchy(`*(Shape)arg1*[, *(bool)names=True*]`)` → list :

Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

`property dispIndex`

Return class index of this instance.

`getVolume(`*(Box)arg1*`)` → float :

Returns the shape volume.

updateAttrs(*(Serializable)arg1*, *(dict)arg2*) → None :
Update object attributes from given dictionary

State

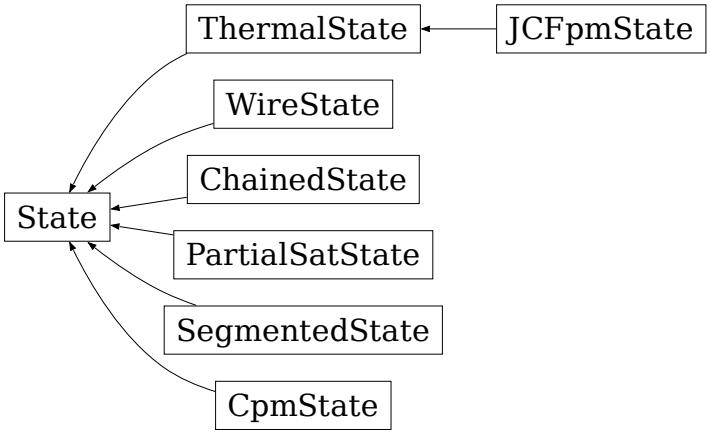


Fig. 20: Inheritance graph of State. See also: *ChainedState*, *CpmState*, *JCFpmState*, *PartialSatState*, *SegmentedState*, *ThermalState*, *WireState*.

Material

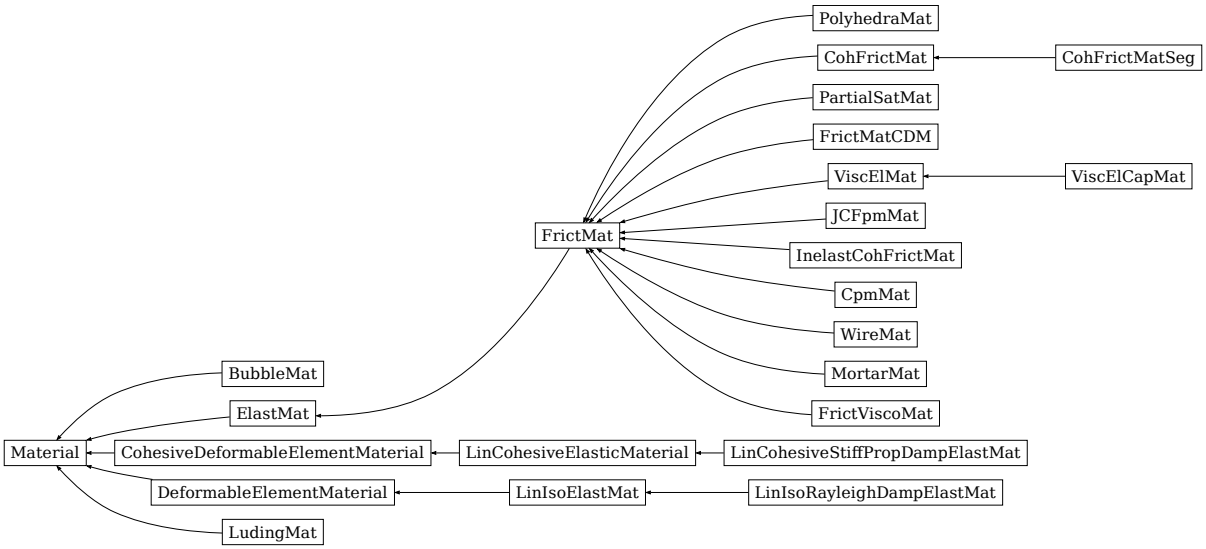


Fig. 21: Inheritance graph of Material. See also: *BubbleMat*, *CohFrictMat*, *CohFrictMatSeg*, *CohesiveDeformableElementMaterial*, *CpmMat*, *DeformableElementMaterial*, *ElastMat*, *FrictMat*, *FrictMatCDM*, *FrictViscoMat*, *InelastCohFrictMat*, *JCFpmMat*, *LinCohesiveElasticMaterial*, *LinCohesiveStiffPropDampElastMat*, *LinIsoElastMat*, *LinIsoRayleighDampElastMat*, *LudingMat*, *MortarMat*, *PartialSatMat*, *PolyhedraMat*, *ViscElCapMat*, *ViscElMat*, *WireMat*.

Bound

2.3.2 Interactions

Interaction

IGeom

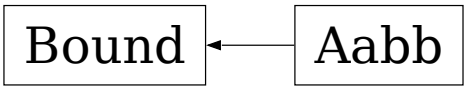


Fig. 22: Inheritance graph of Bound. See also: *Aabb*.

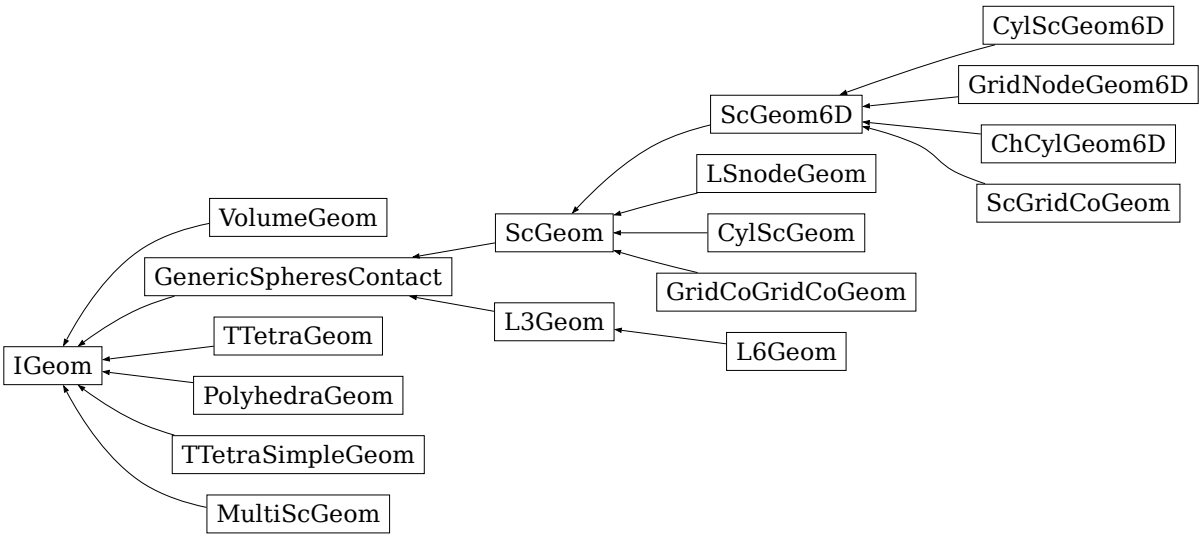


Fig. 23: Inheritance graph of IGeom. See also: *ChCylGeom6D*, *CylScGeom*, *CylScGeom6D*, *GenericSpheresContact*, *GridCoGridCoGeom*, *GridNodeGeom6D*, *L3Geom*, *L6Geom*, *LSnodeGeom*, *MultiScGeom*, *PolyhedraGeom*, *ScGeom*, *ScGeom6D*, *ScGridCoGeom*, *TTetraGeom*, *TTetraSimpleGeom*, *VolumeGeom*.

IPhys

2.3.3 Global engines

GlobalEngine

PeriodicEngine

BoundaryController

Collider

FieldApplier

2.3.4 Partial engines

2.3.5 Dispatchers

2.3.6 Functors

2.3.7 Bounding volume creation

BoundFunctor

BoundDispatcher

2.3.8 Interaction Geometry creation

IGeomFunctor

IGeomDispatcher

2.3.9 Interaction Physics creation

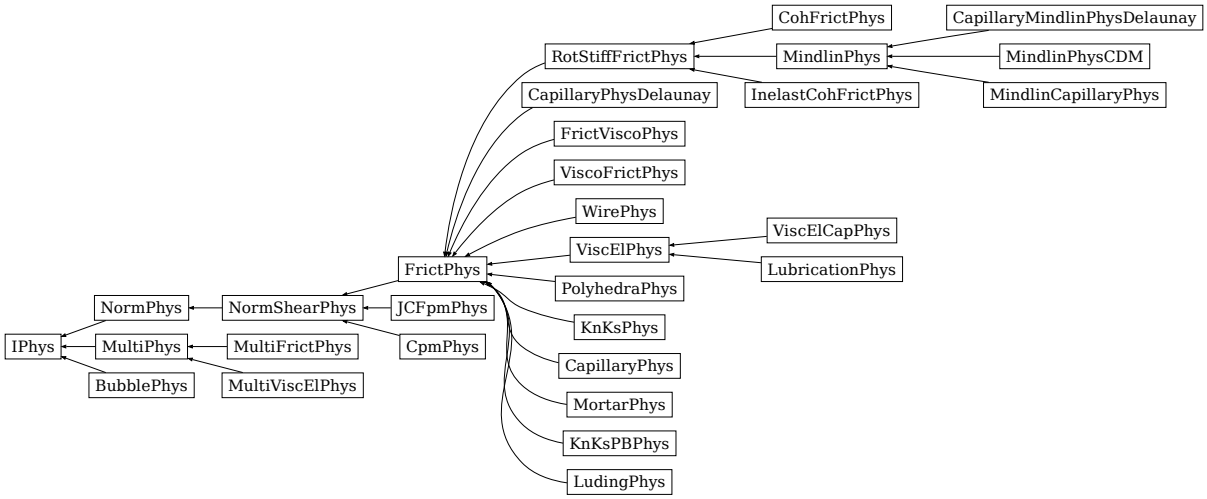


Fig. 24: Inheritance graph of IPhys. See also: *BubblePhys*, *CapillaryMindlinPhysDelaunay*, *CapillaryPhys*, *CapillaryPhysDelaunay*, *CohFrictPhys*, *CpmPhys*, *FrictPhys*, *FrictViscoPhys*, *InelastCohFrictPhys*, *JCFpmPhys*, *KnKsPBPhys*, *KnKsPhys*, *LubricationPhys*, *LudingPhys*, *MindlinCapillaryPhys*, *MindlinPhys*, *MindlinPhysCDM*, *MortarPhys*, *MultiFrictPhys*, *MultiPhys*, *MultiViscElPhys*, *NormPhys*, *NormShearPhys*, *PolyhedraPhys*, *RotStiffFrictPhys*, *ViscElCapPhys*, *ViscElPhys*, *ViscoFrictPhys*, *WirePhys*.

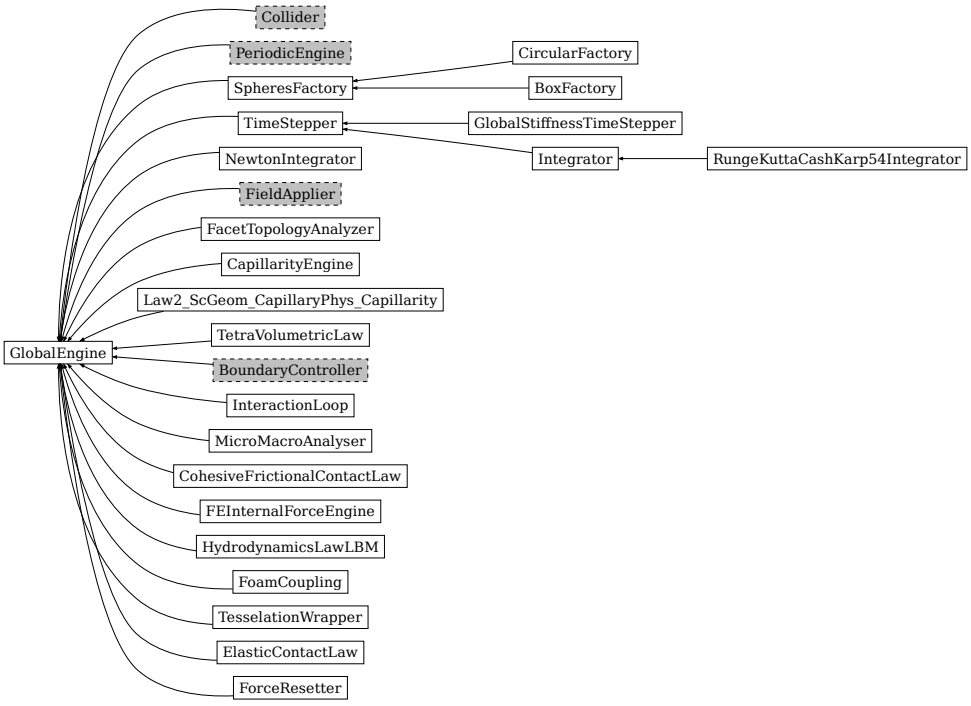


Fig. 25: Inheritance graph of GlobalEngine, gray dashed classes are discussed in their own sections: *Collider*, *PeriodicEngine*, *FieldApplier*, *BoundaryController*. See also: *BoxFactory*, *CapillarityEngine*, *CircularFactory*, *CohesiveFrictionalContactLaw*, *ElasticContactLaw*, *FEInternalForceEngine*, *FacetTopologyAnalyzer*, *FoamCoupling*, *ForceResetter*, *GlobalStiffnessTimeStepper*, *HydrodynamicsLawLBM*, *Integrator*, *InteractionLoop*, *Law2_ScGeom_CapillaryPhys_Capillarity*, *MicroMacroAnalyser*, *NewtonIntegrator*, *RungeKuttaCashKarp54Integrator*, *SpheresFactory*, *TesselationWrapper*, *TetraVolumetricLaw*, *TimeStepper*.

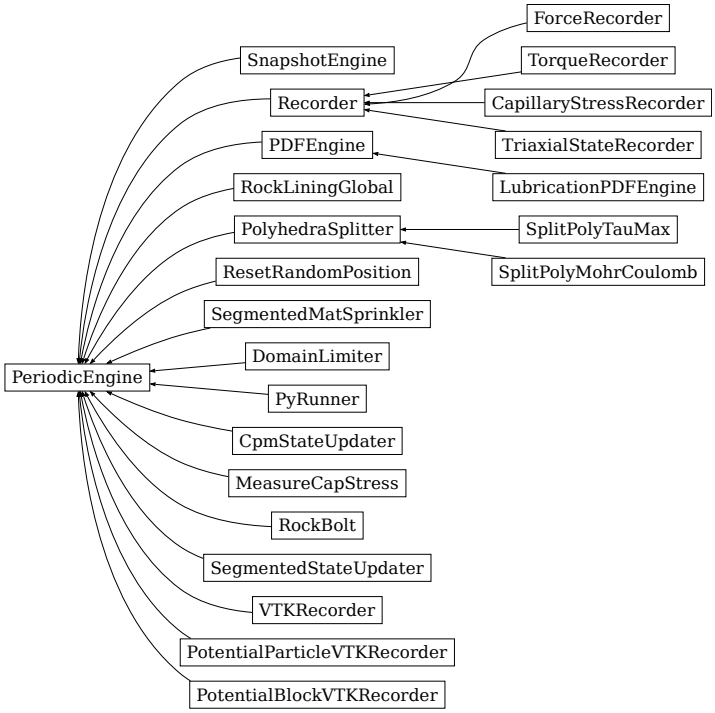


Fig. 26: Inheritance graph of PeriodicEngine. See also: *CapillaryStressRecorder*, *CpmStateUpdater*, *DomainLimiter*, *ForceRecorder*, *LubricationPDFEngine*, *MeasureCapStress*, *PDFEngine*, *PolyhedraSplitter*, *PotentialBlockVTKRecorder*, *PotentialParticleVTKRecorder*, *PyRunner*, *Recorder*, *ResetRandomPosition*, *RockBolt*, *RockLiningGlobal*, *SegmentedMatSprinkler*, *SegmentedStateUpdater*, *SnapshotEngine*, *SplitPolyMohrCoulomb*, *SplitPolyTauMax*, *TorqueRecorder*, *TriaxialStateRecorder*, *VTKRecorder*.

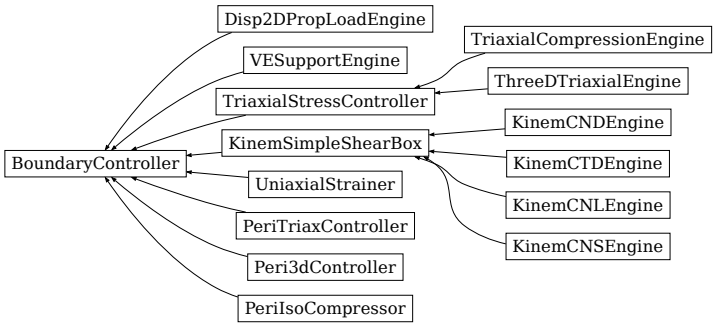


Fig. 27: Inheritance graph of BoundaryController. See also: *Disp2DPropLoadEngine*, *KinemCNDEngine*, *KinemCNLEngine*, *KinemCNSEngine*, *KinemCTDEngine*, *KinemSimpleShearBox*, *Peri3dController*, *PeriIsoCompressor*, *PeriTriaxController*, *ThreeDTriaxialEngine*, *TriaxialCompressionEngine*, *TriaxialStressController*, *UniaxialStrainer*, *VESupportEngine*.

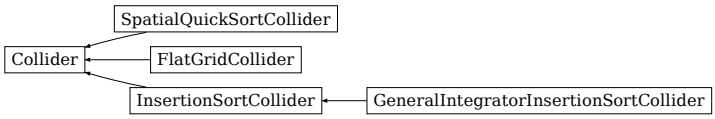


Fig. 28: Inheritance graph of Collider. See also: *FlatGridCollider*, *GeneralIntegratorInsertionSortCollider*, *InsertionSortCollider*, *SpatialQuickSortCollider*.

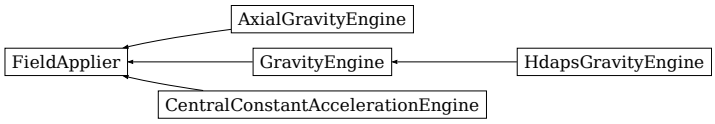


Fig. 29: Inheritance graph of FieldApplier. See also: *AxialGravityEngine*, *CentralConstantAccelerationEngine*, *GravityEngine*, *HdapsGravityEngine*.

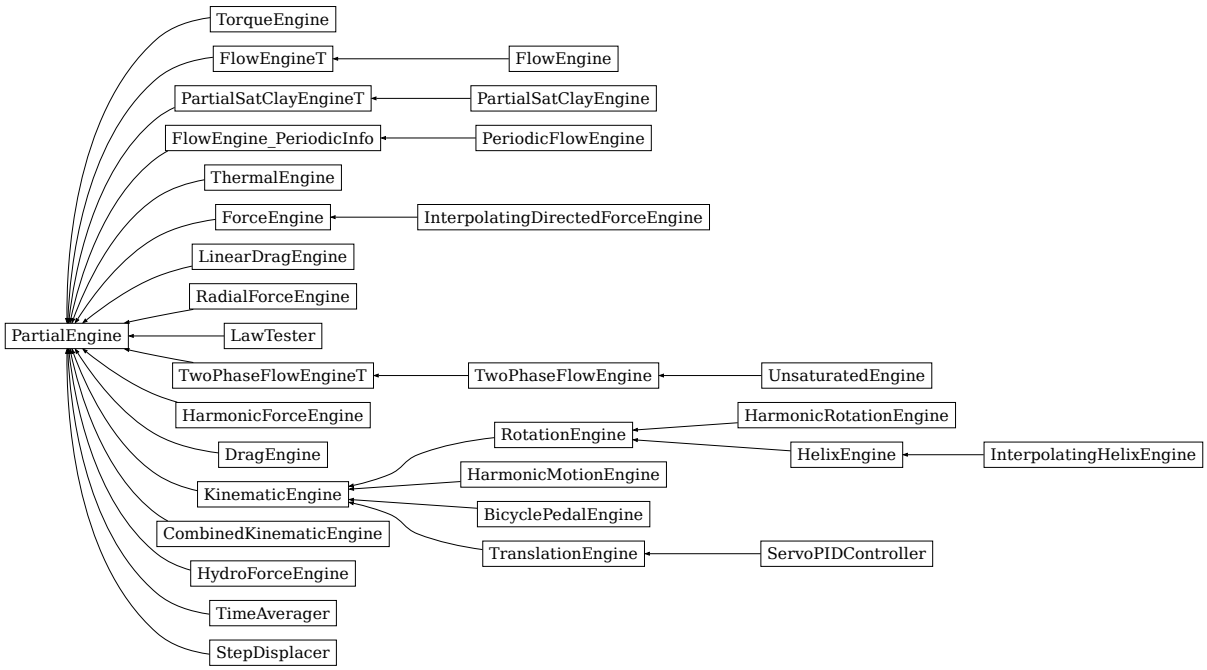


Fig. 30: Inheritance graph of PartialEngine. See also: *BicyclePedalEngine*, *CombinedKinematicEngine*, *DragEngine*, *FlowEngine*, *FlowEngineT*, *FlowEngine_PeriodicInfo*, *ForceEngine*, *HarmonicForceEngine*, *HarmonicMotionEngine*, *HarmonicRotationEngine*, *HelixEngine*, *HydroForceEngine*, *InterpolatingDirectedForceEngine*, *InterpolatingHelixEngine*, *KinematicEngine*, *LawTester*, *LinearDragEngine*, *PartialSatClayEngine*, *PartialSatClayEngineT*, *PeriodicFlowEngine*, *RadialForceEngine*, *RotationEngine*, *ServoPIDController*, *StepDisplacer*, *ThermalEngine*, *TimeAverager*, *TorqueEngine*, *TranslationEngine*, *TwoPhaseFlowEngine*, *TwoPhaseFlowEngineT*, *UnsaturatedEngine*.

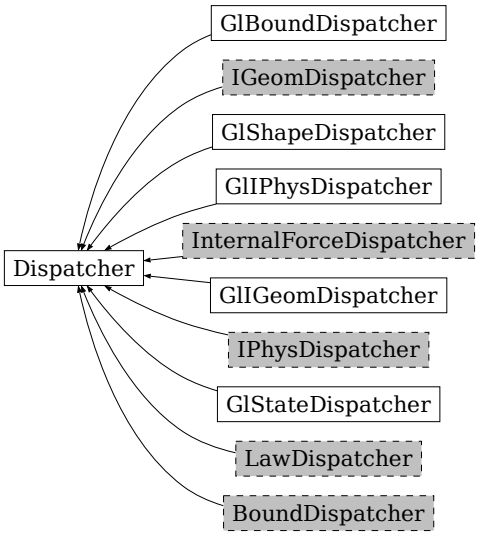


Fig. 31: Inheritance graph of Dispatcher, gray dashed classes are discussed in their own sections: *IGeomDispatcher*, *InternalForceDispatcher*, *IPhysDispatcher*, *LawDispatcher*, *BoundDispatcher*. See also: *GlBoundDispatcher*, *GlIGeomDispatcher*, *GlIPhysDispatcher*, *GlShapeDispatcher*, *GlStateDispatcher*.

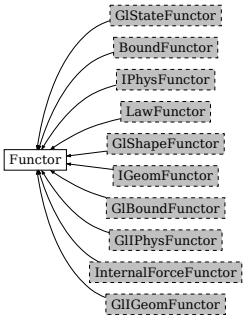


Fig. 32: Inheritance graph of Functor, gray dashed classes are discussed in their own sections: *GlStateFunctor*, *BoundFunctor*, *IPhysFunctor*, *LawFunctor*, *GlShapeFunctor*, *IGeomFunctor*, *GlBoundFunctor*, *GlIPhysFunctor*, *InternalForceFunctor*, *GlIGeomFunctor*.

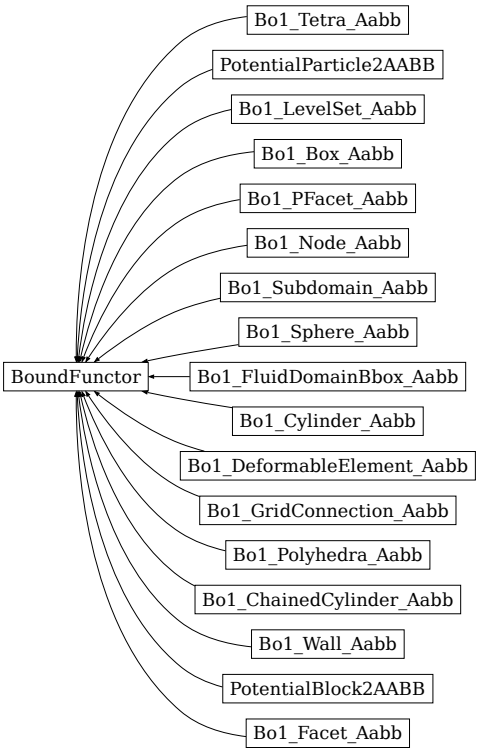


Fig. 33: Inheritance graph of BoundFunctor. See also: *Bo1_Box_Aabb*, *Bo1_ChainedCylinder_Aabb*, *Bo1_Cylinder_Aabb*, *Bo1_DeformableElement_Aabb*, *Bo1_Facet_Aabb*, *Bo1_FluidDomainBbox_Aabb*, *Bo1_GridConnection_Aabb*, *Bo1_LevelSet_Aabb*, *Bo1_Node_Aabb*, *Bo1_PFacet_Aabb*, *Bo1_Polyhedra_Aabb*, *Bo1_Sphere_Aabb*, *Bo1_Subdomain_Aabb*, *Bo1_Tetra_Aabb*, *Bo1_Wall_Aabb*, *PotentialBlock2AABB*, *PotentialParticle2AABB*.

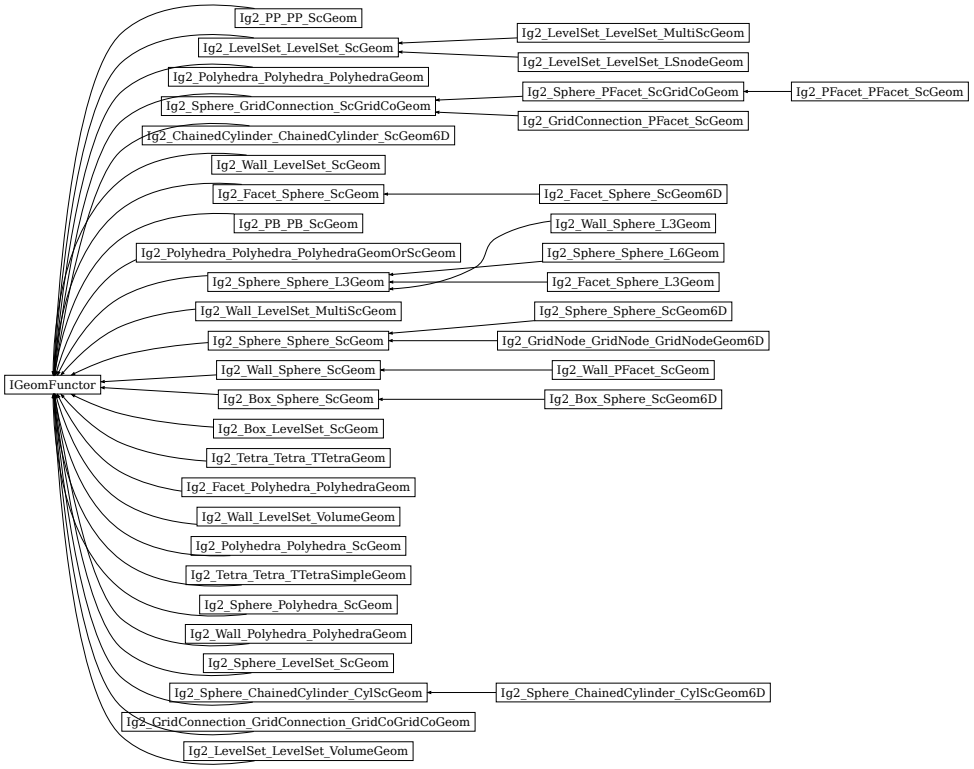


Fig. 34: Inheritance graph of IGeomFunctor. See also: *Ig2_Box_LevelSet_ScGeom*, *Ig2_Box_Sphere_ScGeom*, *Ig2_Box_Sphere_ScGeom6D*, *Ig2_ChainedCylinder_ChainedCylinder_ScGeom6D*, *Ig2_Facet_Polyhedra_PolyhedraGeom*, *Ig2_Facet_Sphere_L3Geom*, *Ig2_Facet_Sphere_ScGeom*, *Ig2_Facet_Sphere_ScGeom6D*, *Ig2_GridConnection_GridConnection_GridCoGridCoGeom*, *Ig2_GridConnection_PFacet_ScGeom*, *Ig2_GridNode_GridNode_GridNodeGeom6D*, *Ig2_LevelSet_LevelSet_LSnodeGeom*, *Ig2_LevelSet_LevelSet_MultiScGeom*, *Ig2_LevelSet_LevelSet_ScGeom*, *Ig2_LevelSet_LevelSet_VolumeGeom*, *Ig2_PB_PB_ScGeom*, *Ig2_PFacet_PFacet_ScGeom*, *Ig2_PP_PP_ScGeom*, *Ig2_Polyhedra_Polyhedra_PolyhedraGeom*, *Ig2_Polyhedra_Polyhedra_PolyhedraGeomOrScGeom*, *Ig2_Polyhedra_Polyhedra_ScGeom*, *Ig2_Sphere_ChainedCylinder_CylScGeom*, *Ig2_Sphere_ChainedCylinder_CylScGeom6D*, *Ig2_Sphere_GridConnection_ScGridCoGeom*, *Ig2_Sphere_LevelSet_ScGeom*, *Ig2_Sphere_PFacet_ScGridCoGeom*, *Ig2_Sphere_Polyhedra_ScGeom*, *Ig2_Sphere_Sphere_L3Geom*, *Ig2_Sphere_Sphere_L6Geom*, *Ig2_Sphere_Sphere_ScGeom*, *Ig2_Sphere_Sphere_ScGeom6D*, *Ig2_Tetra_Tetra_TTetraGeom*, *Ig2_Tetra_Tetra_TTetraSimpleGeom*, *Ig2_Wall_LevelSet_MultiScGeom*, *Ig2_Wall_LevelSet_ScGeom*, *Ig2_Wall_LevelSet_VolumeGeom*, *Ig2_Wall_PFacet_ScGeom*, *Ig2_Wall_Polyhedra_PolyhedraGeom*, *Ig2_Wall_Sphere_L3Geom*, *Ig2_Wall_Sphere_ScGeom*.

IPhysFunctor

Fig. 35: Inheritance graph of IPhysFunctor. See also: *Ip2__2xInelastCohFrictMat__InelastCohFrictPhys*, *Ip2__BubbleMat__BubbleMat__BubblePhys*, *Ip2__CohFrictMatSeg__CohFrictMatSeg__CohFrictPhys*, *Ip2__CohFrictMat__CohFrictMat__CohFrictPhys*, *Ip2__CpmMat__CpmMat__CpmPhys*, *Ip2__ElastMat__ElastMat__NormPhys*, *Ip2__ElastMat__ElastMat__NormShearPhys*, *Ip2__FrictMatCDM__FrictMatCDM__MindlinPhysCDM*, *Ip2__FrictMat__CpmMat__FrictPhys*, *Ip2__FrictMat__FrictMatCDM__MindlinPhysCDM*, *Ip2__FrictMat__FrictMat__CapillaryMindlinPhysDelaunay*, *Ip2__FrictMat__FrictMat__CapillaryPhys*, *Ip2__FrictMat__FrictMat__CapillaryPhysDelaunay*, *Ip2__FrictMat__FrictMat__FrictPhys*, *Ip2__FrictMat__FrictMat__KnKsPBPhys*, *Ip2__FrictMat__FrictMat__KnKsPhys*, *Ip2__FrictMat__FrictMat__LubricationPhys*, *Ip2__FrictMat__FrictMat__MindlinCapillaryPhys*, *Ip2__FrictMat__FrictMat__MindlinPhys*, *Ip2__FrictMat__FrictMat__MultiFrictPhys*, *Ip2__FrictMat__FrictMat__ViscoFrictPhys*, *Ip2__FrictMat__FrictViscoMat__FrictViscoPhys*, *Ip2__FrictMat__PolyhedraMat__FrictPhys*, *Ip2__FrictViscoMat__FrictViscoMat__FrictViscoPhys*, *Ip2__JCFpmMat__JCFpmMat__JCFpmPhys*, *Ip2__LudingMat__LudingMat__LudingPhys*, *Ip2__MortarMat__MortarMat__MortarPhys*, *Ip2__PartialSatMat__PartialSatMat__MindlinPhys*, *Ip2__PolyhedraMat__PolyhedraMat__PolyhedraPhys*, *Ip2__ViscElCapMat__ViscElCapMat__ViscElCapPhys*, *Ip2__ViscElMat__ViscElMat__MultiViscElPhys*, *Ip2__ViscElMat__ViscElMat__ViscElPhys*, *Ip2__WireMat__WireMat__WirePhys*.

IPhysDispatcher

2.3.10 Constitutive laws

LawFunctor

LawDispatcher

2.3.11 Internal forces

InternalForceFunctor

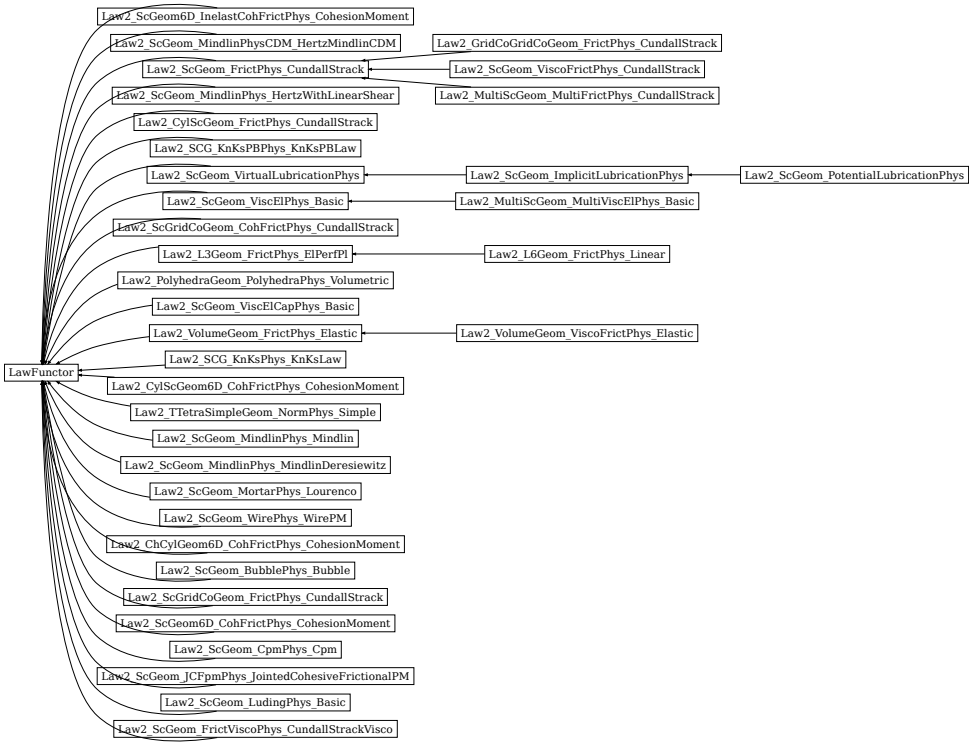


Fig. 36: Inheritance graph of LawFunctor. See also: *Law2_ChCylGeom6D_CohFrictPhys_CohesionMoment*, *Law2_CylScGeom6D_CohFrictPhys_CohesionMoment*, *Law2_CylScGeom_FrictPhys_CundallStrack*, *Law2_GridCoGridCoGeom_FrictPhys_CundallStrack*, *Law2_L3Geom_FrictPhys_ElPerfPl*, *Law2_L6Geom_FrictPhys_Linear*, *Law2_MultiScGeom_MultiFrictPhys_CundallStrack*, *Law2_MultiScGeom_MultiViscElPhys_Basic*, *Law2_PolyhedraGeom_PolyhedraPhys_Volumetric*, *Law2_SCG_KnKsPBPhys_KnKsPBLaw*, *Law2_SCG_KnKsPhys_KnKsLaw*, *Law2_ScGeom6D_CohFrictPhys_CohesionMoment*, *Law2_ScGeom6D_InelastCohFrictPhys_CohesionMoment*, *Law2_ScGeom_BubblePhys_Bubble*, *Law2_ScGeom_CpmPhys_Cpm*, *Law2_ScGeom_FrictPhys_CundallStrack*, *Law2_ScGeom_FrictViscoPhys_CundallStrackVisco*, *Law2_ScGeom_ImplicitLubricationPhys*, *Law2_ScGeom_JCFpmPhys_JointedCohesiveFrictionalPM*, *Law2_ScGeom_LudingPhys_Basic*, *Law2_ScGeom_MindlinPhysCDM_HertzMindlinCDM*, *Law2_ScGeom_MindlinPhys_HertzWithLinearShear*, *Law2_ScGeom_MindlinPhys_Mindlin*, *Law2_ScGeom_MindlinPhys_MindlinDeresiewitz*, *Law2_ScGeom_MortarPhys_Lourenco*, *Law2_ScGeom_PotentialLubricationPhys*, *Law2_ScGeom_VirtualLubricationPhys*, *Law2_ScGeom_ViscElCapPhys_Basic*, *Law2_ScGeom_ViscElPhys_Basic*, *Law2_ScGeom_ViscoFrictPhys_CundallStrack*, *Law2_ScGeom_WirePhys_WirePM*, *Law2_ScGridCoGeom_CohFrictPhys_CundallStrack*, *Law2_ScGridCoGeom_FrictPhys_CundallStrack*, *Law2_TTetraSimpleGeom_NormPhys_Simple*, *Law2_VolumeGeom_FrictPhys_Elastic*, *Law2_VolumeGeom_ViscoFrictPhys_Elastic*.

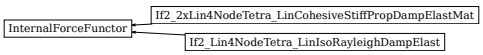


Fig. 37: Inheritance graph of InternalForceFunctor. See also: *If2_2xLin4NodeTetra_LinCohesiveStiffPropDampElastMat*, *If2_Lin4NodeTetra_LinIsoRayleighDampElast*.

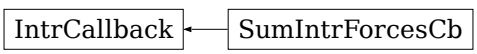


Fig. 38: Inheritance graph of IntrCallback. See also: *SumIntrForcesCb*.

InternalForceDispatcher

2.3.12 Callbacks

2.3.13 Preprocessors

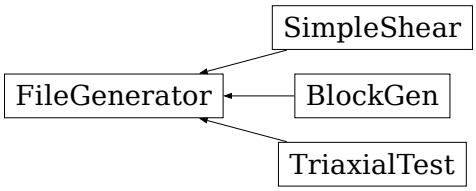


Fig. 39: Inheritance graph of FileGenerator. See also: *BlockGen*, *SimpleShear*, *TriaxialTest*.

2.3.14 Rendering

OpenGLRenderer

GlShapeFunctor

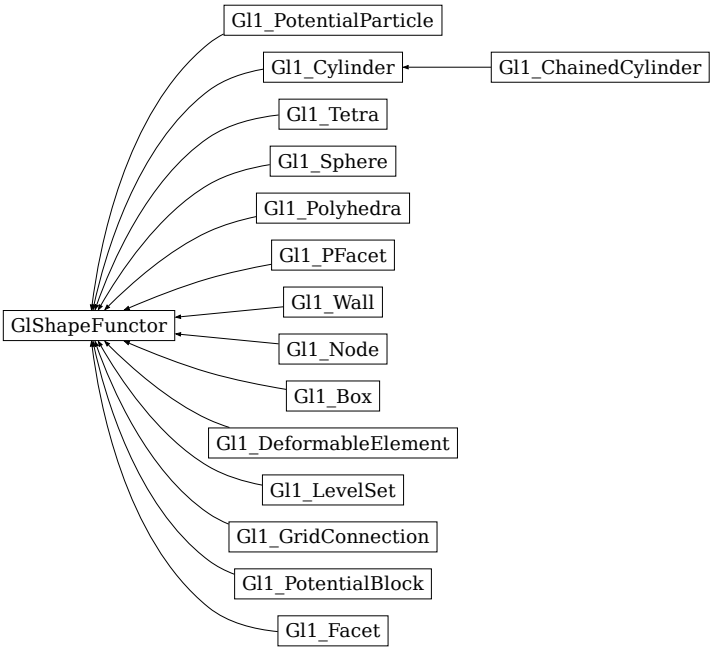


Fig. 40: Inheritance graph of GlShapeFunctor. See also: *Gl1_Box*, *Gl1_ChainedCylinder*, *Gl1_Cylinder*, *Gl1_DeformableElement*, *Gl1_Facet*, *Gl1_GridConnection*, *Gl1_LevelSet*, *Gl1_Node*, *Gl1_PFacet*, *Gl1_Polyhedra*, *Gl1_PotentialBlock*, *Gl1_PotentialParticle*, *Gl1_Sphere*, *Gl1_Tetra*, *Gl1_Wall*.

GlStateFunctor

GlBoundFunctor

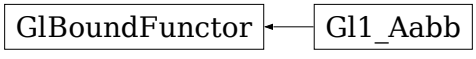


Fig. 41: Inheritance graph of GlBoundFunctor. See also: *Gl1_Aabb*.

GlIGeomFunctor

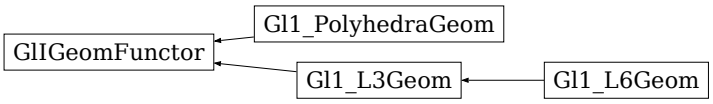


Fig. 42: Inheritance graph of GlIGeomFunctor. See also: *Gl1_L3Geom*, *Gl1_L6Geom*, *Gl1_PolyhedraGeom*.

GlIPhysFunctor

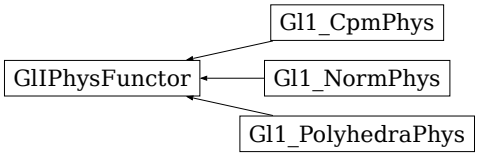


Fig. 43: Inheritance graph of GlIPhysFunctor. See also: *Gl1_CpmPhys*, *Gl1_NormPhys*, *Gl1_PolyhedraPhys*.

2.3.15 Simulation data

Omega

`class yade.wrapper.Omega`

The whole YADE world made of one or, possibly, several *scenes* serving as independent simulations. The Omega instance is accessed as *O*, e.g., *O.bodies*

`addScene(`*(Omega)arg1*`)` → int :

Add new scene to Omega, returns its number

`property bodies`

Bodies in the current simulation (container supporting index access by id and iteration)

`bufferFromIntrsct(`*(Omega)arg1*, *(Subdomain)subdomain*, *(int)rank*, *(int)size*, *(bool)mirror*`)` → object :

returns a (char*) pointer to the underying buffer of intersections[rank], so that it can be overwritten. Size must be passed in advance. Pointer to mirrorIntersections[rank] is returned if mirror=True. Python syntax: bufferFromIntrsct(…)[:]=bytes(something)

`property cell`

Periodic *Cell* of the current scene (None if the scene is aperiodic).

`childClassesNonrecursive(`*(Omega)arg1*, *(str)arg2*`)` → list :

Return list of all classes deriving from given class, as registered in the class factory

`disableGdb(`*(Omega)arg1*`)` → None :

Revert SEGV and ABRT handlers to system defaults.

`property dt`

Current timestep (Δt) value. See *dynDt* for enabling/disabling automatic Δt updates through a *TimeStepper*.

`property dynDt`

Whether a *TimeStepper* (when present in *O.engines*) is used for dynamic Δt control.

`property dynDtAvailable`

Whether a *TimeStepper* is amongst *O.engines*, activated or not.

property energy

EnergyTracker of the current simulation. (meaningful only with *O.trackEnergy*)

property engines

List of engines in the simulation (corresponds to Scene::engines in C++ source code).

exitNoBacktrace(*(Omega)arg1*[, *(int)status=0*]) → None :

Disable SEGV handler and exit, optionally with given status number.

property filename

Filename under which the current simulation was saved (None if never saved).

property forceSyncCount

Counter for number of syncs in ForceContainer, for profiling purposes.

property forces

ForceContainer (forces, torques) in the current simulation.

property interactions

Access to *interactions* of simulation, by using

1. id's of both *Bodies* of the interactions, e.g. `O.interactions[23,65]`
2. iteration over the whole container:

```
for i in O.interactions: print i.id1,i.id2
```

Note

Iteration silently skips interactions that are not *real*.

intrsctToBytes(*(Omega)arg1*, *(Subdomain)subdomain*, *(int)rank*, *(bool)mirror*) → object :

returns a copy of intersections[rank] (a vector<int>) from a subdomain in the form of bytes. Returns a copy mirrorIntersections[rank] if mirror=True.

isChildClassOf(*(Omega)arg1*, *(str)arg2*, *(str)arg3*) → bool :

Tells whether the first class derives from the second one (both given as strings).

property iter

Get current step number

labeledEngine(*(Omega)arg1*, *(str)arg2*) → object :

Return instance of engine/functor with the given label. This function shouldn't be called by the user directly; every ehange in O.engines will assign respective global python variables according to labels.

For example:

O.engines=[InsertionSortCollider(label='collider')]

collider.nBins=5 # collider has become a variable after assignment to O.engines automatically

load(*(Omega)arg1*, *(str)file*[, *(bool)quiet=False*]) → None :

Load simulation from file. The file should have been *saved* in the same version of Yade built or compiled with the same features, otherwise compatibility is not guaranteed. Compatibility may also be affected by different versions of external libraries such as Boost

loadTmp(*(Omega)arg1*[, *(str)mark=''*[, *(bool)quiet=False*]]) → None :

Load simulation previously stored in memory by saveTmp. *mark* optionally distinguishes multiple saved simulations

lsTmp(*(Omega)arg1*) → list :

Return list of all memory-saved simulations.

property materials

Shared materials; they can be accessed by id or by label

property miscParams

MiscParams in the simulation (Scene::mistParams), usually used to save serializables that don't fit anywhere else, like GL functors

property numThreads

Get maximum number of threads openMP can use.

pause(*(Omega)arg1*) → None :

Stop simulation execution. (May be called from within the loop, and it will stop after the current step).

property periodic

Get/set whether the current *scene is periodic* or not (True/False).

plugins(*(Omega)arg1*) → list :

Return list of all plugins registered in the class factory.

property realtime

Return clock (human world) time the simulation has been running, in seconds.

reload(*(Omega)arg1*[, *(bool)quiet=False*]) → None :

Reload current simulation

reset(*(Omega)arg1*) → None :

Reset simulations completely (including another scenes!).

resetAllScenes(*(Omega)arg1*) → None :

Reset all scenes.

resetCurrentScene(*(Omega)arg1*) → None :

Reset current scene.

resetThisScene(*(Omega)arg1*) → None :

Reset current scene.

resetTime(*(Omega)arg1*) → None :

Reset simulation time: step number, virtual and real time. (Doesn't touch anything else, including timings).

run(*(Omega)arg1*[, *(int)nSteps=-1*[, *(bool)wait=False*]]) → None :

Run the simulation. *nSteps* how many steps to run, then stop (if positive); *wait* will cause not returning to python until simulation will have stopped.

runEngine(*(Omega)arg1*, *(Engine)arg2*) → None :

Run given engine exactly once; simulation time, step number etc. will not be incremented (use only if you know what you do).

property running

Whether background thread is currently running a simulation.

save(*(Omega)arg1*, *(str)file*[, *(bool)quiet=False*]) → None :

Save current simulation to file (should be .xml or .xml.bz2 or .yade or .yade.gz). .xml files are bigger than .yade, but can be more or less easily (due to their size) opened and edited, e.g. with text editors. .bz2 and .gz correspond both to compressed versions. There are software requirements for successful reloads, see *O.load*.

saveTmp(*(Omega)arg1*[, *(str)mark=''*[, *(bool)quiet=False*]]) → None :

Save simulation to memory (disappears at shutdown), can be loaded later with loadTmp. *mark* optionally distinguishes different memory-saved simulations.

sceneToString(*(Omega)arg1*) → object :

Return the entire scene as a string. Equivalent to using O.save(…) except that the scene goes to a string instead of a file. (see also stringToScene())

property speed

Return current calculation speed [iter/sec].

step(*(Omega)arg1*) → None :

Advance the simulation by one step. Returns after the step will have finished.

property stopAtIter

Get/set number of iteration after which the simulation will stop.

property stopAtTime

Get/set time after which the simulation will stop.

stringToScene(*(Omega)arg1*, *(str)arg2*[, *(str)mark=''*]) → None :

Load simulation from a string passed as argument (see also sceneToString).

property subStep

Get the current subStep number (only meaningful if O.subStepping==True); -1 when outside the loop, otherwise either 0 (O.subStepping==False) or number of engine to be run (O.subStepping==True)

property subStepping

Get/set whether subStepping is active.

switchScene(*(Omega)arg1*) → None :

Switch to alternative simulation (while keeping the old one). Calling the function again switches back to the first one. Note that most variables from the first simulation will still refer to the first simulation even after the switch (e.g. b=O.bodies[4]; O.switchScene(); [b still refers to the body in the first simulation here])

switchToScene(*(Omega)arg1*, *(int)arg2*) → None :

Switch to defined scene. Default scene has number 0, other scenes have to be created by addScene method.

property tags

Tags (string=string dictionary) of the current simulation (container supporting string-index access/assignment)

property thisScene

Return current scene's id.

property time

Return virtual (model world) time of the simulation.

property timingEnabled

Globally enable/disable timing services (see documentation of the *timing module*).

tmpFilename(*(Omega)arg1*) → str :

Return unique name of file in temporary directory which will be deleted when yade exits.

tmpToFile(*(Omega)arg1*, *(str)fileName*[, *(str)mark=''*]) → None :

Save XML of *saveTmp*'d simulation into *fileName*.

tmpToString(*(Omega)arg1*[, *(str)mark=''*]) → str :

Return XML of *saveTmp*'d simulation as string.

property trackEnergy

When energy tracking is enabled or disabled in this simulation.

wait(*(Omega)arg1*) → None :

Don't return until the simulation will have been paused. (Returns immediately if not running).

BodyContainer

class yade.wrapper.BodyContainer

__init__(*(object)arg1*, *(BodyContainer)arg2*) → None

addToClump(*(BodyContainer)arg1*, *(object)arg2*, *(int)arg3*[, *(int)discretization=0*]) → None :

Add body b (or a list of bodies) to an existing clump c. c must be clump and b may not be a clump member of c. Clump masses and inertia are adapted automatically (for details see *clump()*).

See examples/clumps/addToClump-example.py for an example script.

Note

If b is a clump itself, then all members will be added to c and b will be deleted. If b is a clump member of clump d, then all members from d will be added to c and d will be deleted. If you need to add just clump member b, *release* this member from d first.

append(*(BodyContainer)arg1*, *(Body)arg2*) → int :

Append one Body instance, return its id.

append((BodyContainer)arg1, (object)arg2) -> object :

Append list of Body instance, return list of ids

appendClumped(*(BodyContainer)arg1*, *(object)arg2*[, *(int)discretization=0*]) → tuple :

Append given list of bodies as a clump (rigid aggregate); returns a tuple of `(clumpId, [memberId1,memberId2,...])`. Clump masses and inertia are computed automatically depending upon *discretization* (for details see *clump()*).

clear(*(BodyContainer)arg1*) → None :

Remove all bodies (interactions not checked)

clump(*(BodyContainer)arg1*, *(object)arg2*[, *(int)discretization=0*]) → int :

Clump given bodies together (creating a rigid aggregate); returns `clumpId`. A precise definition of clump masses and inertia when clump members overlap requires spherical members together with *discretization*>0 and is achieved in this case by integration/summation over mass points using a regular grid of cells (grid cells length is defined as L_{min}/`discretization`, where L_{min} is the minimum length of an Axis-Aligned Bounding Box. If *discretization*<=0 sum of inertias from members is simply used, which is faster but accurate only for non-overlapping members).

deleteClumpBody(*(BodyContainer)arg1*, *(Body)arg2*) → None :

Erase clump member.

deleteClumpMember(*(BodyContainer)arg1*, *(Body)arg2*, *(Body)arg3*) → None :

Erase clump member.

property enableRedirection

let collider switch to optimized algorithm with body redirection when bodies are erased - true by default

`erase`(*(BodyContainer)arg1*, *(int)arg2*[, *(bool)eraseClumpMembers=0*]) → bool :

Erase body with the given id; all interaction will be deleted by InteractionLoop in the next step. If a clump is erased use *O.bodies.erase(clumpId,True)* to erase the clump AND its members.

`getRoundness`(*(BodyContainer)arg1*[, *(list)excludeList=[]*]) → float :

Returns roundness coefficient RC = R2/R1. R1 is the equivalent sphere radius of a clump. R2 is the minimum radius of a sphere, that imbeds the clump. If just spheres are present RC = 1. If clumps are present 0 < RC < 1. Bodies can be excluded from the calculation by giving a list of ids: *O.bodies.getRoundness([ids])*.

See examples/clumps/replaceByClumps-example.py for an example script.

`insertAtId`(*(BodyContainer)arg1*, *(Body)arg2*, *(int)insertatid*) → int :

Insert a body at theid, (no body should exist in this id)

`releaseFromClump`(*(BodyContainer)arg1*, *(int)arg2*, *(int)arg3*[, *(int)discretization=0*]) → None :

Release body b from clump c. b must be a clump member of c. Clump masses and inertia are adapted automatically (for details see *clump()*).

See examples/clumps/releaseFromClump-example.py for an example script.

Note

If c contains only 2 members b will not be released and a warning will appear. In this case clump c should be *erased*.

`replace`(*(BodyContainer)arg1*, *(object)arg2*) → object

`replaceByClumps`(*(BodyContainer)arg1*, *(list)arg2*, *(object)arg3*[, *(int)discretization=0*]) → list :

Replace spheres by clumps using a list of clump templates and a list of amounts; returns a list of tuples: `[(clumpId1,[memberId1,memberId2,...]),(clumpId2,[memberId1,memberId2,...]),...]`. A new clump will have the same volume as the sphere, that was replaced. Clump masses and inertia are adapted automatically (for details see *clump()*).

O.bodies.replaceByClumps([utils.clumpTemplate([1,1],[.5,.5])] , [.9]) #will replace 90 % of all standalone spheres by 'dyads'

See examples/clumps/replaceByClumps-example.py for an example script.

`subdomainBodies`(*(BodyContainer)arg1*) → object :

id's of bodies with bounds in MPI subdomain

`updateClumpProperties`(*(BodyContainer)arg1*[, *(list)excludeList=[]*[, *(int)discretization=5*]]) → None :

Manually force Yade to update clump properties mass, volume and inertia (for details of 'discretization' value see *clump()*). Can be used, when clumps are modified or erased during a simulation. Clumps can be excluded from the calculation by giving a list of ids: *O.bodies.updateProperties([ids])*.

`property useRedirection`

true if the scene uses up-to-date lists for boundedBodies and realBodies; turned true automatically 1/ after removal of bodies if *enableRedirection=True* , and 2/ in MPI execution. *(auto-updated)*

InteractionContainer

`class yade.wrapper.InteractionContainer`

Access to *interactions* of simulation, by using

1. id's of both *Bodies* of the interactions, e.g. `O.interactions[23,65]`

2. iteraction over the whole container:

```
for i in O.interactions: print i.id1,i.id2
```

Note

Iteration silently skips interactions that are virtual i.e. not *real*.

`__init__`(*(object)arg1*, *(InteractionContainer)arg2*) → None

`all`(*(InteractionContainer)arg1*[, *(bool)onlyReal=False*]) → list :

Return list of all interactions. Virtual interaction are filtered out if onlyReal=True, else (default) it dumps the full content.

`clear`(*(InteractionContainer)arg1*) → None :

Remove all interactions, and invalidate persistent collider data (if the collider supports it).

`countReal`(*(InteractionContainer)arg1*) → int :

Return number of interactions that are *real*.

`erase`(*(InteractionContainer)arg1*, *(int)arg2*, *(int)arg3*) → None :

Erase one interaction, given by id1, id2 (internally, `requestErase` is called – the interaction might still exist as potential, if the *Collider* decides so).

`eraseNonReal`(*(InteractionContainer)arg1*) → None :

Erase all interactions that are not *real* .

`has`(*(InteractionContainer)arg1*, *(int)id1*, *(int)id2*[, *(bool)onlyReal=False*]) → bool :

Tell if a pair of ids *id1*, *id2* corresponds to an existing interaction (*real* or not depending on *onlyReal*)

`nth`(*(InteractionContainer)arg1*, *(int)arg2*) → Interaction :

Return n-th interaction from the container (usable for picking random interaction). The virtual interactions are not reached.

`property serializeSorted`

None((yade.wrapper.InteractionContainer)arg1) -> bool

`withBody`(*(InteractionContainer)arg1*, *(int)arg2*) → list :

Return list of *real* interactions of given body.

`withBodyAll`(*(InteractionContainer)arg1*, *(int)arg2*) → list :

Return list of all (*real* as well as non-real) interactions of given body.

ForceContainer

`class yade.wrapper.ForceContainer`

`__init__`(*(object)arg1*, *(ForceContainer)arg2*) → None

`addF`(*(ForceContainer)arg1*, *(int)id*, *(yade._minieigenHP.Vector3)f*[, *(bool)permanent=False*]) → None :

Apply force on body (accumulates). The force applies for one iteration, then it is reset by ForceResetter. # permanent parameter is deprecated, instead of addF(…,permanent=True) use setPermF(…).

`addT`(*(ForceContainer)arg1*, *(int)id*, *(yade._minieigenHP.Vector3)t*[, *(bool)permanent=False*]) → None :

Apply torque on body (accumulates). The torque applies for one iteration, then it is reset by ForceResetter. # permanent parameter is deprecated, instead of addT(…,permanent=True) use setPermT(…).

f(*(ForceContainer)arg1*, *(int)id*[, *(bool)sync=False*]) → yade._minieigenHP.Vector3 :

Resultant force on body, excluding *gravity*. For clumps in openMP, synchronize the force container with sync=True, else the value will be wrong.

getPermForceUsed(*(ForceContainer)arg1*) → bool :

Check wether permanent forces are present.

m(*(ForceContainer)arg1*, *(int)id*[, *(bool)sync=False*]) → yade._minieigenHP.Vector3 :

Deprecated alias for t (torque).

permF(*(ForceContainer)arg1*, *(int)id*) → yade._minieigenHP.Vector3 :

read the value of permanent force on body (set with setPermF()).

permT(*(ForceContainer)arg1*, *(int)id*) → yade._minieigenHP.Vector3 :

read the value of permanent torque on body (set with setPermT()).

reset(*(ForceContainer)arg1*[, *(bool)resetAll=True*]) → None :

Reset the force container, including user defined permanent forces/torques. resetAll=False will keep permanent forces/torques unchanged.

setPermF(*(ForceContainer)arg1*, *(int)arg2*, *(yade._minieigenHP.Vector3)arg3*) → None :

set the value of permanent force on body.

setPermT(*(ForceContainer)arg1*, *(int)arg2*, *(yade._minieigenHP.Vector3)arg3*) → None :

set the value of permanent torque on body.

property syncCount

Number of synchronizations of ForceContainer (cummulative); if significantly higher than number of steps, there might be unnecessary syncs hurting performance.

t(*(ForceContainer)arg1*, *(int)id*[, *(bool)sync=False*]) → yade._minieigenHP.Vector3 :

Torque applied on body. For clumps in openMP, synchronize the force container with sync=True, else the value will be wrong.

MaterialContainer

class yade.wrapper.MaterialContainer

Container for *Materials*. A material can be accessed using

1. numerical index in range(0,len(cont)), like cont[2];
2. textual label that was given to the material, like cont['steel']. This entails traversing all materials and should not be used frequently.

__init__(*(object)arg1*, *(MaterialContainer)arg2*) → None

append(*(MaterialContainer)arg1*, *(Material)arg2*) → int :

Add new shared *Material*; changes its id and return it.

append((MaterialContainer)arg1, (object)arg2) -> object :

Append list of *Material* instances, return list of ids.

index(*(MaterialContainer)arg1*, *(str)arg2*) → int :

Return id of material, given its label.

Scene

Cell

2.3.16 Other classes

`class yade.wrapper.Serializable`

`dict`(*(Serializable)arg1*) → dict :

Return dictionary of attributes.

`updateAttrs`(*(Serializable)arg1*, *(dict)arg2*) → None :

Update object attributes from given dictionary

`class yade.wrapper.TimingDeltas`

`property data`

Get timing data as list of tuples (label, execTime[nsec], execCount) (one tuple per checkpoint)

`reset`(*(TimingDeltas)arg1*) → None :

Reset timing information

2.4 Yade modules reference

2.4.1 yade.bf module

Overview

This module contains breakage functions (bf) that can be used for particle breakage by replacement approach. Functions can be used for both spheres and clumps of spheres. However, this module is particularly useful for clumps because it deals with multiple clump-specific issues:

- Clump members do not interact. Hence, modification of the Love-Webber stress tensor is proposed to mimic interactions between clump members when the stress state is computed.
- If clumped spheres overlap, their total mass and volume are bigger than the mass and volume of the clump. Thus, clump should not split by simply releasing clump members. The mass of new sub-particles is adjusted to balance the mass of a nonoverlapping volume of the broken clump member.
- New sub-particles can be generated beyond the outline of the broken clump member to avoid excessive overlapping. Particles are generated taking into account the positions of neighbor particles and additional constraints (e.g. predicate can be prescribed to make sure that new particles are generated inside the container).

Clump breakage algorithm

The typical workflow consists of the following steps (full description in [Brzezinski2022]):

- Stress computation of each clump member. The stress is computed using the Love-Weber (LV) definition of the stress tensor. Then, a proposed correction of the stress tensor is applied.
- Based on the adopted strength criterion, the level of effort for each clump member is computed. Clump breaks if the level of effort for any member is greater than one. Only the most strained member can be split in one iteration.
- The most strained member of the clump is first released from the clump and erased from simulation. New mass and moment of inertia are computed for the new clump. The difference between the "old" and the "new" mass must be replaced by new bodies in the simulation.
- New, smaller spheres are added to the simulation balancing the mass difference. The spheres are placed in the void space, hence do not overlap with other bodies that are already present in the simulation (*splitting_clump*).

- Finally, the soundness of the remaining part of the original clump needs to be verified. If the clump members do not contact each other anymore, the clump needs to be replaced with smaller clumps/spheres (*handling_new_clumps*).
- Optionally, overlapping between new sub-particles of sub-particles and existing bodies can be allowed (*packing_parameters*).

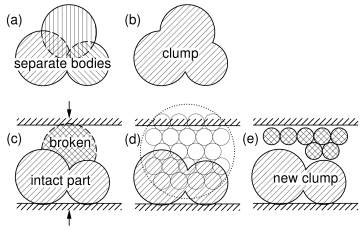


Fig. 44: Stages of creating a clump in Yade software and splitting due to the proposed algorithm: (a) overlapping bodies, (b) clumped body (reduced mass and moments of inertia), (c) selection of clump member for splitting, (d) searching for potential positions of sub-particles, (e) replacing clump member with sub-particles, updating clump mass and moments of inertia.

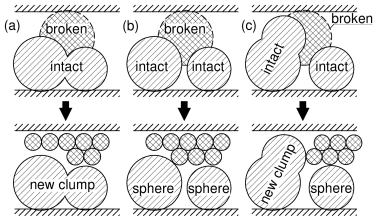


Fig. 45: Different scenarios of clump splitting: (a) clump remains in the simulation - only updated, (b) clump is split into spheres, (c) clump is split into a sphere and a new clump.

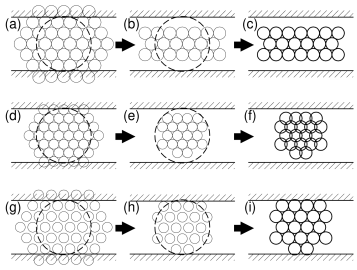


Fig. 46: Replacing sphere with sub-particles: (a-c) non-overlapping, (d-f) overlapping sub-particles and potentially overlapping with neighbor bodies, (g-i) non-overlapping sub-particles but potentially overlapping with neighbor bodies.

Functions required for clump breakage algorithm described in:

Brzeziński, K., & Gladky, A. (2022), Clump breakage algorithm for DEM simulation of crushable aggregates. [Brzezinski2022]

Strength Criterion adopted from;

Gladkyy, A., & Kuna, M. (2017). DEM simulation of polyhedral particle cracking using a combined Mohr–Coulomb–Weibull failure criterion. Granular Matter, 19(3), 41. [Gladky2017]

Source code file

`yade.bf.checkFailure`(*b*, *tension_strength*, *compressive_strength*, *wei_V0=0.01*, *wei_P=0.63*, *wei_m=3*, *weibull=True*, *stress_correction=True*)

Strength criterion adopted from [Gladkyy and Kuna 2017]. Returns particles 'effort' (equivalent stress / strength) if the strength is exceeded, and zero othervise.

yade.bf.**evalClump**(*clump_id*, *radius_ratio*, *tension_strength*, *compressive_strength*, *relative_gap=0*, *wei_V0=0.001*, *wei_P=0.63*, *wei_m=3*, *weibull=True*, *stress_correction=True*, *initial_packing_scale=1.5*, *max_scale=3*, *search_for_neighbours=True*, *outer_predicate=None*, *discretization=20*, *grow_radius=1.0*, *max_grow_radius=2.0*)

Iterates over clump members with "checkFailure" function. Replaces the broken clump member with subparticles. Split new clump if necessary. If clump is not broken returns False, if broken True.

yade.bf.**replaceSphere**(*sphere_id*, *subparticles_mass=None*, *radius_ratio=2*, *relative_gap=0*, *neighbours_ids=[]*, *initial_packing_scale=1.5*, *max_scale=3*, *scale_multiplier=None*, *search_for_neighbours=True*, *outer_predicate=None*, *grow_radius=1.0*, *max_grow_radius=2.0*)

This function is intended to replace sphere with subparticles. It is dedicated for spheres replaced from clumps (but not only). Thus, two features are utilized: - subparticles_mass (mass of the subparticles after replacement), since in a clump only a fraction of original spheres mass is taken into account - neighbours_ids - list of ids of the neighbour bodies (e.g. other clump members, other bodies that sphere is contacting with) that we do not want to penetrate with new spheres (maybe it could be later use to avoid penetration of other bodies). However, passing neighbours_ids is not always necessary. By default (if search_for_neighbours==True), existing spheres are detected authomatically and added to neighbours_ids. Also, outer_predicate can beused to avoid penetrating other bodies with subparticles. Spheres will be initially populated in a hexagonal packing (predicate with dimension of sphere diameter multiplied by initial_packing_scale). Initial packing scale is greater than one to make sure that sufficient number of spheres will be produced to obtain reguired mass (taking into account porosity). scale_multiplier - if a sufficient number of particles cannot be produced with initial packing scale, it is multiplied by scale multiplier. The procedure is repeated until initial_packing scale is reached. If scale_multiplier is None it will be changed to max_scale/initial_packing_scale, so the maximum range will be achieved in second iteration. max_scale - limits the initial_packing_scale which can be increased by the algorithm. If initial_packing_scale >max_scale, sphere will not be replaced (broken). outer_predicate - it is an additional constraint for subparticles generation. Can be used when non spherical bodies are in vicinity of the broken particle, particles are in box etc. search_for_neighbours - if True searches for additional neighbours (spheres whithin a range of initial_packing_scale *sphere_radius)

Particles can be generated with smaller radius and than sligtly growed (by "grow_radius"). It allows for adding extra potential energy in the simulation, and increase the chances for successful packing. relative_gap - is the gap between packed subparticles (relalive to the radius of subparticle), note that if grow_radius > 1, during subparticles arangemenr their radius is temporarily decreased by 1/grow radius. It can be used to create special cases for overlapping (described in the paper).

yade.bf.**stressTensor**(*b*, *stress_correction=True*)

Modification of Love-Weber stress tensor, that applied to the clump members gives results similar to standalone bodies.

2.4.2 yade.bodiesHandling module

Miscellaneous functions, which are useful for handling bodies.

yade.bodiesHandling.**facetsDimensions**(*idFacets=[]*, *mask=-1*)

The function accepts the list of facet id's or list of facets and calculates max and min dimensions, geometrical center.

Parameters

- **idFacets** (*list*) – list of spheres
- **mask** (*int*) – *Body.mask* for the checked bodies

Returns

dictionary with keys `min` (minimal dimension, Vector3), `max` (maximal dimension, Vector3), `minId` (minimal dimension facet Id, Vector3), `maxId` (maximal dimension

facet Id, Vector3), `center` (central point of bounding box, Vector3), `extends` (sizes of bounding box, Vector3), `number` (number of facets, int),

`yade.bodiesHandling.sphereDuplicate(`*idSphere*`)`

The functions makes a copy of sphere

`yade.bodiesHandling.spheresModify(`*idSpheres=[]*, *mask=-1*, *shift=Vector3(0, 0, 0)*, *scale=1.0*, *orientation=Quaternion((1, 0, 0), 0)*, *copy=False*`)`

The function accepts the list of spheres id's or list of bodies and modifies them: rotating, scaling, shifting. if copy=True copies bodies and modifies them. Also the mask can be given. If idSpheres not empty, the function affects only bodies, where the mask passes. If idSpheres is empty, the function search for bodies, where the mask passes.

Parameters

- **shift** (Vector3) – Vector3(X,Y,Z) parameter moves spheres.
- **scale** (*float*) – factor scales given spheres.
- **orientation** (Quaternion) – orientation of spheres
- **mask** (*int*) – *Body.mask* for the checked bodies

Returns

list of bodies if copy=True, and Boolean value if copy=False

`yade.bodiesHandling.spheresPackDimensions(`*idSpheres=[]*, *mask=-1*`)`

The function accepts the list of spheres id's or list of bodies and calculates max and min dimensions, geometrical center.

Parameters

- **idSpheres** (*list*) – list of spheres
- **mask** (*int*) – *Body.mask* for the checked bodies

Returns

dictionary with keys `min` (minimal dimension, Vector3), `max` (maximal dimension, Vector3), `minId` (minimal dimension sphere Id, Vector3), `maxId` (maximal dimension sphere Id, Vector3), `center` (central point of bounding box, Vector3), `extends` (sizes of bounding box, Vector3), `volume` (volume of spheres, Real), `mass` (mass of spheres, Real), `number` (number of spheres, int),

2.4.3 yade.export module

Export (not only) geometry to various formats.

`class yade.export.VTKExporter(`*inherits object*`)`

Class for exporting data to VTK Simple Legacy File (for example if, for some reason, you are not able to use *VTKRecorder*). Supported export of:

- spheres
- facets
- polyhedra
- PotentialBlocks
- interactions
- contact points
- periodic cell

Usage:

- create object `vtkExporter = VTKExporter('baseFileName')`,

- add to `O.engines` a `PyRunner` with `command='vtkExporter.exportSomething(...)'`
- alternatively, just use `vtkExporter.exportSomething(...)` at the end of the script for instance

Example: examples/test/vtk-exporter/vtkExporter.py, examples/test/unv-read/unvReadVTKExport.py.

Parameters

- **baseName** (*string*) – name of the exported files. The files would be named, e.g., `baseName-spheres-snapNb.vtk` or `baseName-facets-snapNb.vtk`
- **startSnap** (*int*) – the numbering of files will start form `startSnap`

exportContactPoints(*ids='all'*, *what={}*, *useRef={}*, *comment='comment'*, *numLabel=None*)

exports contact points (CPs) and defined properties.

Parameters

- **ids** (*[(int,int)]*) – see *exportInteractions()*
- **what** (*dictionary*) – see *exportInteractions()*
- **useRef** (*bool*) – see *exportInteractions()*
- **comment** (*string*) – comment to add to vtk file
- **numLabel** (*int*) – number of file (e.g. time step), if unspecified, the last used value + 1 will be used

exportFacets(*ids='all'*, *what={}*, *comment='comment'*, *numLabel=None*)

exports facets (positions) and defined properties. Facets are exported with multiplicated nodes

Parameters

- **ids** (*[int]*/*"all"*) – if "all", then export all facets, otherwise only facets from integer list
- **what** (*dictionary*) – see *exportSpheres()*
- **comment** (*string*) – comment to add to vtk file
- **numLabel** (*int*) – number of file (e.g. time step), if unspecified, the last used value + 1 will be used

exportFacetsAsMesh(*ids='all'*, *connectivityTable=None*, *what={}*, *comment='comment'*, *numLabel=None*)

exports facets (positions) and defined properties. Facets are exported as mesh (not with multiplicated nodes). Therefore additional parameters connectivityTable is needed

Parameters

- **ids** (*[int]*/*"all"*) – if "all", then export all facets, otherwise only facets from integer list
- **what** (*dictionary*) – see *exportSpheres()*
- **comment** (*string*) – comment to add to vtk file
- **numLabel** (*int*) – number of file (e.g. time step), if unspecified, the last used value + 1 will be used
- **nodes** (*[(float,float,float)*/`Vector3`*]*) – list of coordinates of nodes
- **connectivityTable** (*[(int,int,int)]*) – list of node ids of individual elements (facets)

`exportInteractions`(*ids='all'*, *what={}*, *verticesWhat={}*, *comment='comment'*, *numLabel=None*, *useRef=False*)

exports interactions and defined properties.

Parameters

- **ids** (*[(int,int)]|"all"*) – if "all", then export all interactions, otherwise only interactions from (int,int) list
- **what** (*dictionary*) – what to export. parameter is a name->command dictionary. Name is string under which it is saved to vtk, command is string to evaluate. Note that the interactions are labeled as i in this function. Scalar, vector and tensor variables are supported. For example, to export the stiffness difference (named as `dStiff`) from a certain value (1e9) you should write: `what=dict(dStiff='i.phys.kn-1e9', ... )`
- **verticesWhat** (*dictionary*) – what to export on connected bodies. Bodies are labeled as `b` (or `b1` and `b2` if you need to treat both bodies differently)
- **comment** (*string*) – comment to add to vtk file
- **numLabel** (*int*) – number of file (e.g. time step), if unspecified, the last used value + 1 will be used
- **useRef** (*bool*) – if False (default), use current position of the bodies for export, use reference position otherwise

`exportPeriodicCell`(*comment='comment'*, *numLabel=None*)

exports the *Cell* geometry for periodic simulations.

Parameters

- **comment** (*string*) – comment to add to vtk file
- **numLabel** (*int*) – number of file (e.g. time step), if unspecified, the last used value + 1 will be used

`exportPolyhedra`(*ids='all'*, *what={}*, *comment='comment'*, *numLabel=None*, *useRef=False*)

Exports polyhedrons and defined properties.

Parameters

- **ids** (*[int] | "all"*) – if "all", then export all polyhedrons, otherwise only polyhedrons from integer list
- **what** (*dictionary*) – which additional quantities (in addition to the positions) to export. parameter is name->command dictionary. Name is string under which it is saved to vtk, command is string to evaluate. Note that the bodies are labeled as b in this function. Scalar, vector and tensor variables are supported. For example, to export velocity (named as particleVelocity) and the distance from point (0,0,0) (named as dist) you should write: `what=dict(particleVelocity='b.state.vel',dist='b.state.pos.norm()', ... )`
- **comment** (*string*) – comment to add to vtk file
- **numLabel** (*int*) – number of file (e.g. time step), if unspecified, the last used value + 1 will be used

`exportPotentialBlocks`(*ids='all'*, *what={}*, *comment='comment'*, *numLabel=None*, *useRef=False*)

Exports Potential Blocks and defined properties.

Parameters

- **ids** (*[int] | "all"*) – if "all", then export all Potential Blocks, otherwise only Potential Blocks from integer list

- **what** (*dictionary*) – which additional quantities (in addition to the positions) to export. parameter is name->command dictionary. Name is string under which it is saved to vtk, command is string to evaluate. Note that the bodies are labeled as b in this function. Scalar, vector and tensor variables are supported. For example, to export velocity (named as particleVelocity) and the distance from point (0,0,0) (named as dist) you should write: `what=dict(particleVelocity='b.state.vel',dist='b.state.pos.norm()', ... )`
- **comment** (*string*) – comment to add to vtk file
- **numLabel** (*int*) – number of file (e.g. time step), if unspecified, the last used value + 1 will be used

exportSpheres(*ids='all'*, *what={}*, *comment='comment'*, *numLabel=None*, *useRef=False*)

exports spheres (positions and radius) and defined properties.

Parameters

- **ids** (*[int]|"all"*) – if "all", then export all spheres, otherwise only spheres from integer list
- **what** (*dictionary*) – which additional quantities (other than the position and the radius) to export. parameter is name->command dictionary. Name is string under which it is save to vtk, command is string to evaluate. Note that the bodies are labeled as b in this function. Scalar, vector and tensor variables are supported. For example, to export velocity (with name particleVelocity) and the distance form point (0,0,0) (named as dist) you should write: `what=dict(particleVelocity='b.state.vel',dist='b.state.pos.norm()', ... )`
- **comment** (*string*) – comment to add to vtk file
- **numLabel** (*int*) – number of file (e.g. time step), if unspecified, the last used value + 1 will be used
- **useRef** (*bool*) – if False (default), use current position of the spheres for export, use reference position otherwise

class yade.export.**VTKWriter**(*inherits object*)

USAGE: create object vtk_writer = VTKWriter('base_file_name'), add to engines PyRunner with command='vtk_writer.snapshot()'

snapshot()

yade.export.**gmshGeo**(*filename*, *comment=''*, *mask=-1*, *accuracy=-1*)

Save spheres in geo-file for the following using in GMSH (http://www.geuz.org/gmsh/doc/texinfo/) program. The spheres can be there meshed.

Parameters

- **filename** (*string*) – the name of the file, where sphere coordinates will be exported.
- **mask** (*int*) – export only spheres with the corresponding mask export only spheres with the corresponding mask
- **accuracy** (*float*) – the accuracy parameter, which will be set for the poinst in geo-file. By default: 1./10. of the minimal sphere diameter.

Returns

number of spheres which were exported.

Return type

int

yade.export.text(*filename*, *mask=-1*)

Save sphere coordinates into a text file; the format of the line is: x y z r. Non-spherical bodies are silently skipped. Example added to examples/regular-sphere-pack/regular-sphere-pack.py

Parameters

- **filename** (*string*) – the name of the file, where sphere coordinates will be exported.
- **mask** (*int*) – export only spheres with the corresponding mask

Returns
number of spheres which were written.

Return type
int

yade.export.text2vtk(*inFileName*, *outFileName*, *comment='comment'*)

Converts text file (created by *export.textExt* function) into vtk file. See examples/test/paraview-spheres-solid-section/export_text.py example

Parameters

- **inFileName** (*str*) – name of input text file
- **outFileName** (*str*) – name of output vtk file
- **comment** (*str*) – optional comment in vtk file

yade.export.text2vtkSection(*inFileName*, *outFileName*, *point*, *normal=(1, 0, 0)*)

Converts section through spheres from text file (created by *export.textExt* function) into vtk file. See examples/test/paraview-spheres-solid-section/export_text.py example

Parameters

- **inFileName** (*str*) – name of input text file
- **outFileName** (*str*) – name of output vtk file
- **point** (Vector3|*(float,float,float)*) – coordinates of a point lying on the section plane
- **normal** (Vector3|*(float,float,float)*) – normal vector of the section plane

yade.export.textClumps(*filename*, *format='x_y_z_r_clumpId'*, *comment=''*, *mask=-1*)

Save clumps-members into a text file. Non-clumps members are bodies are silently skipped.

Parameters

- **filename** (*string*) – the name of the file, where sphere coordinates will be exported.
- **comment** (*string*) – the text, which will be added as a comment at the top of file. If you want to create several lines of text, please use '\n#' for next lines.
- **mask** (*int*) – export only spheres with the corresponding mask export only spheres with the corresponding mask

Returns
number of clumps, number of spheres which were written.

Return type
int

yade.export.textExt(*filename*, *format='x_y_z_r'*, *comment=''*, *mask=-1*, *attrs=[]*)

Save sphere coordinates and other parameters into a text file in specific format. Non-spherical bodies are silently skipped. Users can add here their own specific format, giving meaningful names. The first file row will contain the format name. Be sure to add the same format specification in ymport.textExt.

Parameters

- **filename** (*string*) – the name of the file, where sphere coordinates will be exported.
- **format** (*string*) – the name of output format. Supported 'x_y_z_r'(default), 'x_y_z_r_matId', 'x_y_z_r_attrs' (use proper comment)
- **comment** (*string*) – the text, which will be added as a comment at the top of file. If you want to create several lines of text, please use '\n#' for next lines. With 'x_y_z_r_attrs' format, the last (or only) line should consist of column headers of quantities passed as attrs (1 comment word for scalars, 3 comment words for vectors and 9 comment words for matrices)
- **mask** (*int*) – export only spheres with the corresponding mask export only spheres with the corresponding mask
- **attrs** (*[str]*) – attributes to be exported with 'x_y_z_r_attrs' format. Each str in the list is evaluated for every body exported with body=b (i.e. 'b.state.pos.norm()' would stand for distance of body from coordinate system origin)

Returns
number of spheres which were written.

Return type
int

yade.export.**textPolyhedra**(*fileName*, *comment=''*, *mask=-1*, *explanationComment=True*, *attrs=[]*)

Save polyhedra into a text file. Non-polyhedra bodies are silently skipped.

Parameters

- **filename** (*string*) – the name of the output file
- **comment** (*string*) – the text, which will be added as a comment at the top of file. If you want to create several lines of text, please use '\n#' for next lines.
- **mask** (*int*) – export only polyhedra with the corresponding mask
- **explanationComment** (*str*) – inclde explanation of format to the beginning of file

Returns
number of polyhedra which were written.

Return type
int

2.4.4 yade.geom module

Creates geometry objects from facets.

yade.geom.**facetBox**(*center*, *extents*, *orientation=Quaternion((1, 0, 0), 0)*, *wallMask=63*, ***kw*)

Create arbitrarily-aligned box composed of facets, with given center, extents and orientation. If any of the box dimensions is zero, corresponding facets will not be created. The facets are oriented outwards from the box.

Parameters

- **center** (Vector3) – center of the box
- **extents** (Vector3) – half lengths of the box sides
- **orientation** (Quaternion) – orientation of the box

- **wallMask** (*bitmask*) – determines which walls will be created, in the order -x (1), +x (2), -y (4), +y (8), -z (16), +z (32). The numbers are ANDed; the default 63 means to create all walls
- ****kw** – (unused keyword arguments) passed to *utils.facet*

Returns
list of facets forming the box

yade.geom.**facetBunker**(*center*, *dBunker*, *dOutput*, *hBunker*, *hOutput*, *hPipe=0.0*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *wallMask=4*, *angleRange=None*, *closeGap=False*, ***kw*)

Create arbitrarily-aligned bunker, composed of facets, with given center, radii, heights and orientation. Return List of facets forming the bunker;

Parameters

- **center** (Vector3) – center of the created bunker
- **dBunker** (*float*) – bunker diameter, top
- **dOutput** (*float*) – bunker output diameter
- **hBunker** (*float*) – bunker height
- **hOutput** (*float*) – bunker output height
- **hPipe** (*float*) – bunker pipe height
- **orientation** (Quaternion) – orientation of the bunker; the reference orientation has axis along the +x axis.
- **segmentsNumber** (*int*) – number of edges on the bunker surface (>=5)
- **wallMask** (*bitmask*) – determines which walls will be created, in the order up (1), down (2), side (4). The numbers are ANDed; the default 7 means to create all walls
- **angleRange** (*(min,θmax)*) – allows one to create only part of bunker by specifying range of angles; if None, (0,2*pi) is assumed.
- **closeGap** (*bool*) – close range skipped in angleRange with triangular facets at cylinder bases.
- ****kw** – (unused keyword arguments) passed to utils.facet;

yade.geom.facetCone(*center*, *radiusTop*, *radiusBottom*, *height*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *wallMask=7*, *angleRange=None*, *closeGap=False*, *radiusTopInner=-1*, *radiusBottomInner=-1*, ***kw*)

Create arbitrarily-aligned cone composed of facets, with given center, radius, height and orientation. Return List of facets forming the cone;

Parameters

- **center** (Vector3) – center of the created cylinder
- **radiusTop** (*float*) – cone top radius
- **radiusBottom** (*float*) – cone bottom radius
- **radiusTopInner** (*float*) – inner radius of cones top, -1 by default
- **radiusBottomInner** (*float*) – inner radius of cones bottom, -1 by default
- **height** (*float*) – cone height
- **orientation** (Quaternion) – orientation of the cone; the reference orientation has axis along the +x axis.
- **segmentsNumber** (*int*) – number of edges on the cone surface (>=5)
- **wallMask** (*bitmask*) – determines which walls will be created, in the order up (1), down (2), side (4). The numbers are ANDed; the default 7 means to create all walls
- **angleRange** (*(min,θmax)*) – allows one to create only part of cone by specifying range of angles; if None, (0,2*pi) is assumed.
- **closeGap** (*bool*) – close range skipped in angleRange with triangular facets at cylinder bases.
- ****kw** – (unused keyword arguments) passed to utils.facet;

yade.geom.facetCylinder(*center*, *radius*, *height*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *wallMask=7*, *angleRange=None*, *closeGap=False*, *radiusTopInner=-1*, *radiusBottomInner=-1*, ***kw*)

Create arbitrarily-aligned cylinder composed of facets, with given center, radius, height and orientation. Return List of facets forming the cylinder;

Parameters

- **center** (Vector3) – center of the created cylinder
- **radius** (*float*) – cylinder radius
- **height** (*float*) – cylinder height
- **radiusTopInner** (*float*) – inner radius of cylinders top, -1 by default
- **radiusBottomInner** (*float*) – inner radius of cylinders bottom, -1 by default
- **orientation** (Quaternion) – orientation of the cylinder; the reference orientation has axis along the +x axis.
- **segmentsNumber** (*int*) – number of edges on the cylinder surface (>=5)
- **wallMask** (*bitmask*) – determines which walls will be created, in the order up (1), down (2), side (4). The numbers are ANDed; the default 7 means to create all walls
- **angleRange** (*(min,θmax)*) – allows one to create only part of bunker by specifying range of angles; if None, (0,2*pi) is assumed.
- **closeGap** (*bool*) – close range skipped in angleRange with triangular facets at cylinder bases.

- ****kw** – (unused keyword arguments) passed to utils.facet;

yade.geom.**facetCylinderConeGenerator**(*center*, *radiusTop*, *height*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *wallMask=7*, *angleRange=None*, *closeGap=False*, *radiusBottom=-1*, *radiusTopInner=-1*, *radiusBottomInner=-1*, ***kw*)

Please, do not use this function directly! Use geom.facetCylinder and geom.facetCone instead. This is the base function for generating cylinders and cones from facets.

Parameters

- **radiusTop** (*float*) – top radius
- **radiusBottom** (*float*) – bottom radius
- ****kw** – (unused keyword arguments) passed to utils.facet;

yade.geom.**facetHelix**(*center*, *radiusOuter*, *pitch*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *angleRange=None*, *radiusInner=0*, ***kw*)

Create arbitrarily-aligned helix composed of facets, with given center, radius (outer and inner), pitch and orientation. Return List of facets forming the helix;

Parameters

- **center** (Vector3) – center of the created cylinder
- **radiusOuter** (*float*) – outer radius
- **radiusInner** (*float*) – inner height (can be 0)
- **orientation** (Quaternion) – orientation of the helix; the reference orientation has axis along the +x axis.
- **segmentsNumber** (*int*) – number of edges on the helix surface (>=3)
- **angleRange** (*(min,θmax)*) – range of angles; if None, (0,2*pi) is assumed.
- ****kw** – (unused keyword arguments) passed to utils.facet;

yade.geom.**facetParallelepiped**(*center*, *extents*, *height*, *orientation=Quaternion((1, 0, 0), 0)*, *wallMask=63*, ***kw*)

Create arbitrarily-aligned Parallelepiped composed of facets, with given center, extents, height and orientation. If any of the parallelepiped dimensions is zero, corresponding facets will not be created. The facets are oriented outwards from the parallelepiped.

Parameters

- **center** (Vector3) – center of the parallelepiped
- **extents** (Vector3) – half lengths of the parallelepiped sides
- **height** (Real) – height of the parallelepiped (along axis z)
- **orientation** (Quaternion) – orientation of the parallelepiped
- **wallMask** (*bitmask*) – determines which walls will be created, in the order -x (1), +x (2), -y (4), +y (8), -z (16), +z (32). The numbers are ANDed; the default 63 means to create all walls
- ****kw** – (unused keyword arguments) passed to *utils.facet*

Returns

list of facets forming the parallelepiped

yade.geom.**facetPolygon**(*center*, *radiusOuter*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *angleRange=None*, *radiusInner=0*, ***kw*)

Create arbitrarily-aligned polygon composed of facets, with given center, radius (outer and inner) and orientation. Return List of facets forming the polygon;

Parameters

- **center** (Vector3) – center of the created cylinder
- **radiusOuter** (*float*) – outer radius
- **radiusInner** (*float*) – inner height (can be 0)
- **orientation** (Quaternion) – orientation of the polygon; the reference orientation has axis along the +x axis.
- **segmentsNumber** (*int*) – number of edges on the polygon surface (>=3)
- **angleRange** (*(min,θmax)*) – allows one to create only part of polygon by specifying range of angles; if None, (0,2*pi) is assumed.
- ****kw** – (unused keyword arguments) passed to utils.facet;

yade.geom.**facetPolygonHelixGenerator**(*center*, *radiusOuter*, *pitch=0*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *angleRange=None*, *radiusInner=0*, ***kw*)

Please, do not use this function directly! Use geom.facetPloygon and geom.facetHelix instead. This is the base function for generating polygons and helixes from facets.

yade.geom.**facetSphere**(*center*, *radius*, *thetaResolution=8*, *phiResolution=8*, *returnElementMap=False*, ***kw*)

Create arbitrarily-aligned sphere composed of facets, with given center, radius and orientation. Return List of facets forming the sphere. Parameters inspired by ParaView sphere glyph

Parameters

- **center** (Vector3) – center of the created sphere
- **radius** (*float*) – sphere radius
- **thetaResolution** (*int*) – number of facets around “equator”
- **phiResolution** (*int*) – number of facets between “poles” + 1
- **returnElementMap** (*bool*) – returns also tuple of nodes ((x1,y1,z1),(x2,y2,z2),…) and elements ((id01,id02,id03),(id11,id12,id13),…) if true, only facets otherwise
- ****kw** – (unused keyword arguments) passed to utils.facet;

2.4.5 yade.gridpfacet module

Helper functions for creating cylinders, grids and membranes. For more details on this type of elements see [Effeindzourou2016], [Effeindzourou2015a], [Bourrier2013],.

For examples using *GridConnections*, see

- examples/grids/CohesiveGridConnectionSphere.py
- examples/grids/GridConnection_Spring.py
- examples/grids/Simple_Grid_Falling.py
- examples/grids/Simple_GridConnection_Falling.py

For examples using *PFacets*, see

- examples/pfacet/gts-pfacet.py
- examples/pfacet/mesh-pfacet.py
- examples/pfacet/pfacetcreators.py

yade.gridpfacet.**chainedCylinder**(*begin=Vector3(0, 0, 0)*, *end=Vector3(1, 0, 0)*, *radius=0.2*, *dynamic=None*, *fixed=False*, *wire=False*, *color=None*, *highlight=False*, *material=-1*, *mask=1*)

Create and connect a chainedCylinder with given parameters. The shape generated by repeted calls of this function is the Minkowski sum of polyline and sphere.

Parameters

- **radius** (Real) – radius of sphere in the Minkowski sum.
- **begin** (Vector3) – first point positioning the line in the Minkowski sum
- **last** (Vector3) – last point positioning the line in the Minkowski sum

In order to build a correct chain, last point of element of rank N must correspond to first point of element of rank N+1 in the same chain (with some tolerance, since bounding boxes will be used to create connections.

Returns

Body object with the *ChainedCylinder shape*.

Note

ChainedCylinder is deprecated and will be removed in the future, use *GridConnection* instead. See *gridpfacet.cylinder* and *gridpfacet.cylinderConnection*.

yade.gridpfacet.**cylinder**(*begin=Vector3(0, 0, 0)*, *end=Vector3(1, 0, 0)*, *radius=0.2*, *nodesIds=[]*, *cylIds=[]*, *dynamic=None*, *fixed=False*, *wire=False*, *color=None*, *highlight=False*, *intMaterial=-1*, *extMaterial=-1*, *mask=1*)

Create a cylinder with given parameters. The shape corresponds to the Minkowski sum of line-segment and sphere, hence, the cylinder has rounded vertices. The cylinder (*GridConnection*) and its corresponding nodes (yref:*GridNodes<GridNode>*) are automatically added to the simulation. The lists with nodes and cylinder ids will be updated automatically.

Parameters

- **begin** (Vector3) – first point of the Minkowski sum in the global coordinate system.
- **end** (Vector3) – last point of the Minkowski sum in the global coordinate system.
- **radius** (Real) – radius of sphere in the Minkowski sum.
- **nodesIds** (*list*) – list with ids of already existing *GridNodes*. New ids will be added.
- **cylIds** (*list*) – list with ids of already existing *GridConnections*. New id will be added.
- **intMaterial** – *Body.material* used to create the interaction physics between the two GridNodes
- **extMaterial** – *Body.material* used to create the interaction physics between the Cylinder (GridConnection) and other bodies (e.g., spheres interaction with the cylinder)

See *utils.sphere*'s documentation for meaning of other parameters.

yade.gridpfacet.**cylinderConnection**(*vertices*, *radius=0.2*, *nodesIds=[]*, *cylIds=[]*, *dynamic=None*, *fixed=False*, *wire=False*, *color=None*, *highlight=False*, *intMaterial=-1*, *extMaterial=-1*, *mask=1*)

Create a chain of cylinders with given parameters. The cylinders (*GridConnection*) and its corresponding nodes (yref:*GridNodes<GridNode>*) are automatically added to the simulation. The lists with nodes and cylinder ids will be updated automatically.

Parameters

vertices (*[Vector3]*) – coordinates of vertices to connect in the global coordinate system.

See *gridpfacet.cylinder* documentation for meaning of other parameters.

yade.gridpfacet.**gmshPFacet**(*meshfile='file.mesh'*, *shift=Vector3(0, 0, 0)*, *scale=1.0*, *orientation=Quaternion((1, 0, 0), 0)*, *radius=1.0*, *wire=True*, *fixed=True*, *materialNodes=-1*, *material=-1*, *color=None*)

Imports mesh geometry from .mesh file and automatically creates connected PFacet elements. For an example see examples/pfacet/mesh-pfacet.py.

Parameters

- **filename** (*string*) – .gts file to read.
- **shift** (*[float,float,float]*) – [X,Y,Z] parameter shifts the mesh.
- **scale** (*float*) – factor scales the mesh.
- **orientation** (*quaternion*) – orientation of the imported geometry.
- **radius** (*float*) – radius used to create the *PFacets*.
- **materialNodes** – specify *Body.material* of *GridNodes*. This material is used to make the internal connections.
- **material** – specify *Body.material* of *PFacets*. This material is used for interactions with external bodies.

See documentation of *utils.sphere* for meaning of other parameters.

Returns

lists of *GridNode* ids *nodesIds*, *GridConnection* ids *cylIds*, and *PFacet* ids *pfIds*

mesh files can easily be created with GMSH.

Additional examples of mesh-files can be downloaded from http://www-roc.inria.fr/gamma/download/download.php

yade.gridpfacet.**gridConnection**(*id1*, *id2*, *radius*, *wire=False*, *color=None*, *highlight=False*, *material=-1*, *mask=1*, *cellDist=None*)

Create a *GridConnection* by connecting two *GridNodes*.

Parameters

- **id1,id2** – the two *GridNodes* forming the cylinder.
- **radius** (*float*) – radius of the cylinder. Note that the radius needs to be the same as the one for the *GridNodes*.
- **cellDist** (Vector3) – for periodic boundary conditions, see *Interaction.cellDist*. Note: periodic boundary conditions for gridConnections are not yet implemented!

See documentation of *utils.sphere* for meaning of other parameters.

Returns

Body object with the *GridConnection shape*.

Note

The material of the *GridNodes* will be used to set the constitutive behaviour of the internal connection, i.e., the constitutive behaviour of the cylinder. The material of the *GridConnection* is used for interactions with other (external) bodies.

yade.gridpfacet.**gridNode**(*center*, *radius*, *dynamic=None*, *fixed=False*, *wire=False*, *color=None*, *highlight=False*, *material=-1*)

Create a *GridNode* which is needed to set up *GridConnections*.

See documentation of *utils.sphere* for meaning of parameters.

Returns
Body object with the *gridNode shape*.

yade.gridpfacet.**gtsPFacet**(*meshfile*, *shift=Vector3(0, 0, 0)*, *scale=1.0*, *radius=1*, *wire=True*, *fixed=True*, *materialNodes=-1*, *material=-1*, *color=None*)

Imports mesh geometry from .gts file and automatically creates connected *PFacet3* elements. For an example see examples/pfacet/gts-pfacet.py.

Parameters

- **filename** (*string*) – .gts file to read.
- **shift** (*[float,float,float]*) – [X,Y,Z] parameter shifts the mesh.
- **scale** (*float*) – factor scales the mesh.
- **radius** (*float*) – radius used to create the *PFacets*.
- **materialNodes** – specify *Body.material* of *GridNodes*. This material is used to make the internal connections.
- **material** – specify *Body.material* of *PFacets*. This material is used for interactions with external bodies.

See documentation of *utils.sphere* for meaning of other parameters.

Returns
lists of *GridNode* ids *nodesIds*, *GridConnection* ids *cylIds*, and *PFacet* ids *pfIds*

yade.gridpfacet.**pfacet**(*id1*, *id2*, *id3*, *wire=True*, *color=None*, *highlight=False*, *material=-1*, *mask=1*, *cellDist=None*)

Create a *PFacet* element from 3 *GridNodes* which are already connected via 3 *GridConnections*:

Parameters

- **id1,id2,id3** – already with *GridConnections* connected *GridNodes*
- **wire** (*bool*) – if `True`, top and bottom facet are shown as skeleton; otherwise facets are filled.
- **color** (*Vector3-or-None*) – color of the PFacet; random color will be assigned if `None`.
- **cellDist** (`Vector3`) – for periodic boundary conditions, see *Interaction.cellDist*. Note: periodic boundary conditions are not yet implemented for PFacets!

See documentation of *utils.sphere* for meaning of other parameters.

Returns
Body object with the *PFacet shape*.

Note

GridNodes and *GridConnections* need to have the same radius. This is also the radius used to create the *PFacet*

yade.gridpfacet.**pfacetCreator1**(*vertices*, *radius*, *nodesIds=[]*, *cylIds=[]*, *pfIds=[]*, *wire=False*, *fixed=True*, *materialNodes=-1*, *material=-1*, *color=None*)

Create a *PFacet* element from 3 vertices and automatically append to simulation. The function uses the vertices to create *GridNodes* and automatically checks for existing nodes.

Parameters

- **vertices** (*[Vector3,Vector3,Vector3]*) – coordinates of vertices in the global coordinate system.
- **radius** (*float*) – radius used to create the *PFacets*.
- **nodesIds** (*list*) – list with ids of already existing *GridNodes*. New ids will be added.
- **cylIds** (*list*) – list with ids of already existing *GridConnections*. New ids will be added.
- **pfIds** (*list*) – list with ids of already existing *PFacets*. New ids will be added.
- **materialNodes** – specify *Body.material* of *GridNodes*. This material is used to make the internal connections.
- **material** – specify *Body.material* of *PFacets*. This material is used for interactions with external bodies.

See documentation of *utils.sphere* for meaning of other parameters.

yade.gridpfacet.**pfacetCreator2**(*id1*, *id2*, *vertex*, *radius*, *nodesIds=[]*, *wire=True*, *materialNodes=-1*, *material=-1*, *color=None*, *fixed=True*)

Create a *PFacet* element from 2 already existing and connected *GridNodes* and one vertex. The element is automatically appended to the simulation.

Parameters

- **id1,id2** (*int*) – ids of already with *GridConnection* connected *GridNodes*.
- **vertex** (Vector3) – coordinates of the vertex in the global coordinate system.

See documentation of *gridpfacet.pfacetCreator1* for meaning of other parameters.

yade.gridpfacet.**pfacetCreator3**(*id1*, *id2*, *id3*, *cylIds=[]*, *pfIds=[]*, *wire=True*, *material=-1*, *color=None*, *fixed=True*, *mask=-1*)

Create a *PFacet* element from 3 already existing *GridNodes* which are not yet connected. The element is automatically appended to the simulation.

Parameters

id1,id2,id3 (*int*) – id of the 3 *GridNodes* forming the *PFacet*.

See documentation of *gridpfacet.pfacetCreator1* for meaning of other parameters.

yade.gridpfacet.**pfacetCreator4**(*id1*, *id2*, *id3*, *pfIds=[]*, *wire=True*, *material=-1*, *color=None*, *fixed=True*, *mask=-1*)

Create a *PFacet* element from 3 already existing *GridConnections*. The element is automatically appended to the simulation.

Parameters

id1,id2,id3 (*int*) – id of the 3 *GridConnections* forming the *PFacet*.

See documentation of *gridpfacet.pfacetCreator1* for meaning of other parameters.

2.4.6 yade.libVersions module

The `yade.libVersions` module tracks versions of all libraries it was compiled with. Example usage is as follows:

```
from yade.libVersions import *
if(getVersion('cgal') > (4,9,0)):
        …
else:
        …
```

To obtain a list of all libraries use the function *libVersions.printAllVersions*.

All libraries listed in *prerequisites* are detected by this module.

Note

If we need a version of some library not listed in *prerequisites*, then it must also be added to that list.

When adding a new version please have a look at these three files:

1. py/_libVersions.cpp: detection of versions from `#include` files by C++.
2. py/libVersions.py.in: python module which is constructed by cmake during compilation. All `*.in` files are processed by cmake.
3. cMake/FindMissingVersions.cmake: forced detection of library with undetectable version.

Hint

The safest way to compare versions is to use builtin python tuple comparison e.g. `if(cgalVer > (4,9,0) and cgalVer < (5,1,1)):`.

`yade.libVersions.getAllVersions`(*rstFormat=False*)

Returns
`str` - this function returns the result of *printAllVersions(rstFormat)* call inside a string variable.

`yade.libVersions.getAllVersionsCmake`()

This function returns library versions as provided by cmake during compilation.

Returns
dictionary in following format: `{ "libName" : [ (major, minor, patchlevel) , "versionString" ] }`

As an example the dict below reflects what libraries this documentation was compiled with (here are only those detected by CMAKE):

```
Yade [1]: from yade.libVersions import *

Yade [2]: getAllVersionsCmake()
Out[2]:
{'cmake': [(4, 3, 4), '4.3.4'],
 'compiler': [(16, 2, 0), '/usr/bin/x86_64-linux-gnu-g++ 16.2.0'],
 'boost': [(1, 90, 0), '1.90.0'],
 'qt': [(5, 15, 19), '5.15.19'],
 'freeglut': [(3, 0, 0), '3.0.0'],
 'eigen': [(3, 5, 0), '3.5.0'],
 'vtk': [(9, 6, 2), '9.6.2'],
 'suitesparse': [(7, 14, 1), '7.14.1'],
 'mpi': [(5, 0, 0), '5.0'],
 'numpy': [(2, 4, 6), '2.4.6'],
 'ipython': [(9, 11, 0), '9.11.0'],
 'sphinx': [(9, 1, 0), '9.1.0'],
 'clp': [(1, 17, 10), '1.17.10'],
 'coinutils': [(2, 11, 12), '2.11.12'],
 'mpi4py': [(4, 1, 2), '4.1.2'],
 'mpmath': [(1, 4, 1), '1.4.1'],
 'tkinter': [(8, 6, 0), '8.6'],
```

(continues on next page)

(continued from previous page)

```
'pygraphviz': [(1, 14, 0), '1.14'],
'Xlib': [(0, 33, 0), '(0,33)']}
```

Note

Please add here detection of other libraries when yade starts using them or if you discover how to extract from cmake a version which I didn't add here.

`yade.libVersions.getArchitecture()`

Returns

string containing processor architecture name, as reported by `uname -m` call or from `CMAKE_HOST_SYSTEM_PROCESSOR` cmake variable.

`yade.libVersions.getLinuxVersion()`

Returns

string containing linux release and version, preferably the value of `PRETTY_NAME` from file `/etc/os-release`.

`yade.libVersions.getVersion(`*libName*`)`

This function returns the tuple `(major, minor, patchlevel)` with library version number. The `yade --test` in file py/tests/libVersions.py tests that this version is the same as detected by cmake and C++. If only one of those could detect the library version, then this number is used.

Parameters

libName (*string*) – the name of the library

Returns

tuple in format `(major, minor, patchlevel)` if `libName` exists. Otherwise it returns `None`.

Note

library openblas has no properly defined version in header files, this function will return `(0,0,0)` for openblas. Parsing the version string would be unreliable. The mpi version detected by cmake sometimes is different than version detected by C++, this needs further investigation.

`yade.libVersions.printAllVersions(`*rstFormat=False*`)`

This function prints a nicely formatted table with library versions.

Parameters

rstFormat (*bool*) – whether to print table using the reStructuredText formatting. Defaults to `False` and prints using Gitlab markdown rules so that it is easy to paste into gitlab discussions.

As an example the table below actually reflects with what libraries this documentation was compiled:

```
Yade [1]: printAllVersions()

```
Yade version    :  20261020-9630~a198ad4~forky1
Yade features   :  LOGGER USEFUL_ERRORS COMPLEX_MP VTK OPENMP GTS QT5 CGAL␣
↪PFVFLOW PFVFLOW LINSOLV MPI TWOPHASEFLOW LS_DEM FEMLIKE GL2PS LBMFLOW THERMAL␣
↪PARTIALSAT POTENTIAL_PARTICLES POTENTIAL_BLOCKS
Yade config dir:  ~/.config/yadedaily
```

(continues on next page)
```

(continued from previous page)

```
Yade precision :  53 bits, 15 decimal places, with mpmath, PrecisionDouble
Yade RealHP<…> :  (15, 33, 45, 60, 120, 150, 300) decimal digits in C++, (15,␣
↪33) decimal digits accessible from python
```

Libraries used :

library	cmake	C++
Xlib	(0,33)	
boost	1.90.0	1.90.0
cgal		6.2.1
clp	1.17.10	1.17.10
cmake	4.3.4	
coinutils	2.11.12	2.11.12
compiler	/usr/bin/x86_64-linux-gnu-g++ 16.2.0	gcc 16.2.0
eigen	3.5.0	3.5.0
freeglut	3.0.0	
gl		20220530
ipython	9.11.0	
metis		unknown_version
mpi	5.0	ompi:6.0.0
mpi4py	4.1.2	
mpmath	1.4.1	
numpy	2.4.6	
openblas		OpenBLAS 0.3.34
pygraphviz	1.14	
python		3.14.8
qglviewer		2.8.0
qt	5.15.19	5.15.19
sphinx	9.1.0	
sqlite		3.53.4
suitesparse	7.14.1	7.14.1
tkinter	8.6	
vtk	9.6.2	9.6.2

```
Linux version  :  Debian GNU/Linux forky/sid
Architecture   :  amd64
Little endian  :  True
```

Note

For convenience at startup `from yade.libVersions import printAllVersions` is executed, so that this function is readily accessible.

`yade._libVersions.getAllVersionsCpp()` → dict

This function returns library versions as discovered by C++ during compilation from all the `#include` headers. This can be useful in debugging to detect some library `.so` conflicts.

Returns

dictionary in folowing format: `{ "libName" : [ (major, minor, patch) , "versionString" ] }`

As an example the dict below reflects what libraries this documentation was compiled with (here

are only those detected by C++):

```
Yade [1]: from yade.libVersions import *

Yade [2]: getAllVersionsCpp()
Out[2]:
{'compiler': [(16, 2, 0), 'gcc 16.2.0'],
 'boost': [(1, 90, 0), '1.90.0'],
 'qt': [(5, 15, 19), '5.15.19'],
 'gl': [(2022, 5, 30), '20220530'],
 'qglviewer': [(2, 8, 0), '2.8.0'],
 'python': [(3, 14, 8), '3.14.8'],
 'eigen': [(3, 5, 0), '3.5.0'],
 'sqlite': [(3, 53, 4), '3.53.4'],
 'vtk': [(9, 6, 2), '9.6.2'],
 'cgal': [(6, 2, 1), '6.2.1'],
 'suitesparse': [(7, 14, 1), '7.14.1'],
 'openblas': [(0, 0, 0), ' OpenBLAS 0.3.34 '],
 'metis': [(0, 0, 0), 'unknown_version'],
 'mpi': [(6, 0, 0), 'ompi:6.0.0'],
 'clp': [(1, 17, 10), '1.17.10'],
 'coinutils': [(2, 11, 12), '2.11.12'],
 'mpfr': [],
 'mpc': []}
```

Note

Please add here C++ detection of other libraries when yade starts using them.

2.4.7 yade.linterpolation module

Module for rudimentary support of manipulation with piecewise-linear functions (which are usually interpolations of higher-order functions, whence the module name). Interpolation is always given as two lists of the same length, where the x-list must be increasing.

Periodicity is supported by supposing that the interpolation can wrap from the last x-value to the first x-value (which should be 0 for meaningful results).

Non-periodic interpolation can be converted to periodic one by padding the interpolation with constant head and tail using the sanitizeInterpolation function.

There is a c++ template function for interpolating on such sequences in pkg/common/Engine/PartialEngine/LinearInterpolate.hpp (stateful, therefore fast for sequential reads).

TODO: Interpolating from within python is not (yet) supported.

`yade.linterpolation.integral`(*x*, *y*)

Return integral of piecewise-linear function given by points x0,x1,... and y0,y1,...

`yade.linterpolation.revIntegrateLinear`(*I*, *x0*, *y0*, *x1*, *y1*)

Helper function, returns value of integral variable x for linear function f passing through (x0,y0),(x1,y1) such that 1. x [x0,x1] 2. _x0^x f dx=I and raise exception if such number doesn't exist or the solution is not unique (possible?)

`yade.linterpolation.sanitizeInterpolation`(*x*, *y*, *x0*, *x1*)

Extends piecewise-linear function in such way that it spans at least the x0...x1 interval, by adding constant padding at the beginning (using y0) and/or at the end (using y1) or not at all.

yade.linterpolation.**xFractionalFromIntegral**(*integral*, *x*, *y*)

Return x within range x0…xn such that _x0^x f dx==integral. Raises error if the integral value is not reached within the x-range.

yade.linterpolation.**xFromIntegral**(*integralValue*, *x*, *y*)

Return x such that _x0^x f dx==integral. x wraps around at xn. For meaningful results, therefore, x0 should == 0

2.4.8 yade.log module

The `yade.log` module serves as an interface to yade logging framework implemented on top of boost::log. For full documentation see *debugging section*. Example usage in python is as follows:

```
import yade.log
yade.log.setLevel('PeriTriaxController',yade.log.TRACE)
```

Example usage in C++ is as follows:

```
LOG_WARN("Something: "<<something)
```

yade._log.**defaultConfigFileName**() → str

Returns
the default log config file, which is loaded at startup, if it exists.

yade._log.**getAllLevels**() → dict

Returns
A python dictionary with all known loggers in yade. Those without a debug level set will have value -1 to indicate that `Default` filter log level is to be used for them.

yade._log.**getDefaultLogLevel**() → int

Returns
The current `Default` filter log level.

yade._log.**getMaxLevel**() → int

Returns
the MAX_LOG_LEVEL of the current yade build.

yade._log.**getUsedLevels**() → dict

Returns
A python dictionary with all used log levels in yade. Those without a debug level (value -1) are omitted.

yade._log.**readConfigFile**(*(str)arg1*) → None

Loads the given configuration file.

Parameters
fname (*str*) – the config file to be loaded.

yade._log.**resetOutputStream**() → None

Resets log output stream to default state: all logs are printed on `std::clog` channel, which usually redirects to `std::cerr`.

yade._log.**saveConfigFile**(*(str)arg1*) → None

Saves log config to specified file.

Parameters
fname (*str*) – the config file to be saved.

yade._log.setDefaultLogLevel(*(int)arg1*) → None

Parameters

`level` (*int*) – Sets the `Default` filter log level, same as calling `log.setLevel("Default",level)`.

yade._log.setLevel(*(str)arg1*, *(int)arg2*) → None

Set filter level (constants `TRACE` (6), `DEBUG` (5), `INFO` (4), `WARN` (3), `ERROR` (2), `FATAL` (1), `NOFILTER` (0)) for given logger.

Parameters

- `className` (*str*) – The logger name for which the filter level is to be set. Use name `Default` to change the default filter level.
- `level` (*int*) – The filter level to be set.

Warning

setting `Default` log level higher than `MAX_LOG_LEVEL` provided during compilation will have no effect. Logs will not be printed because they are removed during compilation.

yade._log.setOutputStream(*(str)arg1*, *(bool)arg2*) → None

Parameters

- `streamName` (*str*) – sets the output stream, special names `cout`, `cerr`, `clog` use the `std::cout`, `std::cerr`, `std::clog` counterpart (`std::clog` the is the default output stream). Every other name means that log will be written to a file with name provided in the argument.
- `reset` (*bool*) – dictates whether all previously set output streams are to be removed. When set to false: the new output stream is set additionally to the current one.

yade._log.setUseColors(*(bool)arg1*) → None

Turn on/off colors in log messages. By default is on. If logging to a file then it is better to be turned off.

yade._log.testAllLevels() → None

This function prints test messages on all log levels. Can be used to see how filtering works and to what streams the logs are written.

yade._log.testOnceLevels() → None

This function prints test messages on all log levels using LOG_ONCE_* macro family.

yade._log.testTimedLevels() → None

This function prints timed test messages on all log levels. In this test the log levels [0...2] are timed to print every 2 seconds, levels [3,4] every 1 second and levels [5,6] every 0.5 seconds. The loop lasts for 2.1 seconds. Can be used to see how timed filtering works and to what streams the logs are written.

yade._log.unsetLevel(*(str)arg1*) → None

Parameters

`className` (*str*) – The logger name for which the filter level is to be unset, so that a `Default` will be used instead. Unsetting the `Default` level will change it to max level and print everything.

2.4.9 yade.math module

This python module exposes all C++ math functions for Real and Complex types to python. In fact it sort of duplicates `import math`, `import cmath` or `import mpmath`. Also it facilitates migration of old python scripts to high precision calculations.

This module has following purposes:

1. To reliably test all C++ math functions of arbitrary precision Real and Complex types against mpmath.
2. To act as a "migration helper" for python scripts which call python mathematical functions that do not work well with mpmath. As an example see *math.linspace* below and this merge request
3. To allow writing python math code in a way that mirrors C++ math code in Yade. As a bonus it will be faster than mpmath because mpmath is a purely python library (which was one of the main difficulties when writing lib/high-precision/ToFromPythonConverter.hpp)
4. To test Eigen NumTraits
5. To test CGAL NumTraits

If another `C++` *math function* is needed it should be added to following files:

1. lib/high-precision/MathFunctions.hpp
2. py/high-precision/_math.cpp
3. py/tests/testMath.py
4. py/tests/testMathHelper.py

If another `python` math function does not work well with `mpmath` it should be added below, and original calls to this function should call this function instead, e.g. `numpy.linspace(…)` is replaced with `yade.math.linspace(…)`.

The `RealHP<n>` *higher precision* math functions can be accessed in python by using the `.HPn` module scope. For example:

```
import yade.math as mth
mth.HP2.sqrt(2) # produces square root of 2 using RealHP<2> precision
mth.sqrt(2)     # without using HPn module scope it defaults to RealHP<1>
```

`yade.math.Real1`(*arg*)

This function is for compatibility of calls like: `g = yade.math.toHP1("-9.81")`. If yade is compiled with default `Real` precision set as `double`, then python won't accept string arguments as numbers. However when using higher precisions only calls `yade.math.toHP1("1.23456789012345678901234567890123456789")` do not cut to the first 15 decimal places. The calls such as `yade.math.toHP1(1.23456789012345678901234567890123456789)` will use default `python double` conversion and will cut the number to its first 15 digits.

If you are debugging a high precision python script, and have difficulty finding places where such cuts have happened you should use `yade.math.toHP1(string)` for declaring all python floating point numbers which are physically important in the simulation. This function will throw exception if bad conversion is about to take place.

Also see example high precision check checkGravityRungeKuttaCashKarp54.py.

`yade.math.degrees`(*arg*)

Returns
arg in radians converted to degrees, using yade.math.Real precision.

`yade.math.degreesHP1`(*arg*)

Returns
arg in radians converted to degrees, using yade.math.Real precision.

yade.math.eig(*a*)

This function calls `numpy.linalg.eig(…)` or `mpmath.eig(…)`, because `numpy.linalg.eig` function does not work with mpmath.

yade.math.getRealHPCppDigits10()

Returns
tuple containing amount of decimal digits supported on C++ side by Eigen and CGAL.

yade.math.getRealHPPythonDigits10()

Returns
tuple containing amount of decimal digits supported on python side by yade.minieigenHP.

yade.math.linspace(*a*, *b*, *num*)

This function calls `numpy.linspace(…)` or `mpmath.linspace(…)`, because `numpy.linspace` function does not work with mpmath.

yade.math.needsMpmathAtN(*N*)

Parameters
`N` – The `int` N value of `RealHP<N>` in question. Must be `N >= 1`.

Returns
`True` or `False` with information if using `mpmath` is necessary to avoid losing precision when working with `RealHP<N>`.

yade.math.radians(*arg*)

The default python function `import math ; math.radians(arg)` only works on 15 digit `double` precision. If you want to carry on calculations in higher precision it is advisable to use this function `yade.math.radiansHP1(arg)` instead. It uses full yade `Real` precision numbers.

NOTE: in the future this function may replace `radians(…)` function which is called in yade in many scripts, and which in fact is a call to native python `math.radians`. We only need to find the best backward compatible approach for this. The function `yade.math.radiansHP1(arg)` will remain as the function which uses native yade `Real` precision.

yade.math.radiansHP1(*arg*)

The default python function `import math ; math.radians(arg)` only works on 15 digit `double` precision. If you want to carry on calculations in higher precision it is advisable to use this function `yade.math.radiansHP1(arg)` instead. It uses full yade `Real` precision numbers.

NOTE: in the future this function may replace `radians(…)` function which is called in yade in many scripts, and which in fact is a call to native python `math.radians`. We only need to find the best backward compatible approach for this. The function `yade.math.radiansHP1(arg)` will remain as the function which uses native yade `Real` precision.

yade.math.toHP1(*arg*)

This function is for compatibility of calls like: `g = yade.math.toHP1("-9.81")`. If yade is compiled with default `Real` precision set as `double`, then python won't accept string arguments as numbers. However when using higher precisions only calls `yade.math.toHP1("1.23456789012345678901234567890123456789012345678901234567890")` do not cut to the first 15 decimal places. The calls such as `yade.math.toHP1(1.23456789012345678901234567890123456789012345678901234567890)` will use default `python double` conversion and will cut the number to its first 15 digits.

If you are debugging a high precision python script, and have difficulty finding places where such cuts have happened you should use `yade.math.toHP1(string)` for declaring all python floating point numbers which are physically important in the simulation. This function will throw exception if bad conversion is about to take place.

Also see example high precision check checkGravityRungeKuttaCashKarp54.py.

yade.math.usesHP()

Returns
`True` if yade is using default `Real` precision higher than 15 digit (53 bits) `double` type.

yade._math.CGAL_Is_finite(*(float)x*) → bool

CGAL's function `Is_finite`, as described in CGAL algebraic foundations exposed to python for testing of CGAL numerical traits.

Returns
`bool` indicating if the `Real` argument is finite.

yade._math.CGAL_Is_valid(*(float)x*) → bool

CGAL's function `Is_valid`, as described in CGAL algebraic foundations exposed to python for testing of CGAL numerical traits.

Returns
`bool` indicating if the `Real` argument is valid. Checks are performed against NaN and Inf.

yade._math.CGAL_Kth_root(*(int)arg1*, *(float)x*) → float

CGAL's function `Kth_root`, as described in CGAL algebraic foundations exposed to python for testing of CGAL numerical traits.

Returns
`Real` the k-th root of argument.

yade._math.CGAL_Sgn(*(float)x*) → int

CGAL's function `Sgn`, as described in CGAL algebraic foundations exposed to python for testing of CGAL numerical traits.

Returns
sign of the argument, can be `-1`, `0` or `1`. Not very useful in python. In C++ it is useful to obtain a sign of an expression with exact accuracy, CGAL starts using MPFR internally for this when the approximate interval contains zero inside it.

yade._math.CGAL_Sqrt(*(float)x*) → float

CGAL's function `Sqrt`, as described in CGAL algebraic foundations exposed to python for testing of CGAL numerical traits.

Returns
`Real` the square root of argument.

yade._math.CGAL_Square(*(float)x*) → float

CGAL's function `Square`, as described in CGAL algebraic foundations exposed to python for testing of CGAL numerical traits.

Returns
`Real` the argument squared.

yade._math.CGAL_To_interval(*(float)x*) → tuple

CGAL's function `To_interval`, as described in CGAL algebraic foundations exposed to python for testing of CGAL numerical traits.

Returns
`(double,double)` tuple inside which the high-precision `Real` argument resides.

yade._math.CGAL_simpleTest() → float

Tests a simple CGAL calculation. Distance between plane and point, uses CGAL's sqrt and pow.

Returns
3.0

yade._math.Catalan([*(int)Precision=53*]) → float

Returns
`Real` The catalan constant, exposed to python for testing of eigen numerical traits.

yade._math.Euler([*(int)Precision=53*]) → float

Returns
`Real` The Euler–Mascheroni constant, exposed to python for testing of eigen numerical traits.

class yade._math.HP1

AddCost = 1

CGAL_Is_finite(*(float)x*) → bool :

CGAL's function `Is_finite`, as described in CGAL algebraic foundations exposed to python for testing of CGAL numerical traits.

Returns
`bool` indicating if the `Real` argument is finite.

CGAL_Is_valid(*(float)x*) → bool :

CGAL's function `Is_valid`, as described in CGAL algebraic foundations exposed to python for testing of CGAL numerical traits.

Returns
`bool` indicating if the `Real` argument is valid. Checks are performed against NaN and Inf.

CGAL_Kth_root(*(int)arg1*, *(float)x*) → float :

CGAL's function `Kth_root`, as described in CGAL algebraic foundations exposed to python for testing of CGAL numerical traits.

Returns
`Real` the k-th root of argument.

CGAL_Sgn(*(float)x*) → int :

CGAL's function `Sgn`, as described in CGAL algebraic foundations exposed to python for testing of CGAL numerical traits.

Returns
sign of the argument, can be `-1`, `0` or `1`. Not very useful in python. In C++ it is useful to obtain a sign of an expression with exact accuracy, CGAL starts using MPFR internally for this when the approximate interval contains zero inside it.

CGAL_Sqrt(*(float)x*) → float :

CGAL's function `Sqrt`, as described in CGAL algebraic foundations exposed to python for testing of CGAL numerical traits.

Returns
`Real` the square root of argument.

CGAL_Square(*(float)x*) → float :

CGAL's function `Square`, as described in CGAL algebraic foundations exposed to python for testing of CGAL numerical traits.

Returns
`Real` the argument squared.

CGAL_To_interval(*(float)x*) → tuple :

CGAL's function `To_interval`, as described in CGAL algebraic foundations exposed to python for testing of CGAL numerical traits.

Returns
`(double,double)` tuple inside which the high-precision `Real` argument resides.

`CGAL_simpleTest()` → float :

Tests a simple CGAL calculation. Distance between plane and point, uses CGAL's sqrt and pow.

Returns
3.0

`Catalan(`[*(int)Precision=53*]`)` → float :

Returns
`Real` The catalan constant, exposed to python for testing of eigen numerical traits.

`Complex`

alias of `complex`

`ComplexAddCost = 2`

`ComplexMulCost = 6`

`ComplexReadCost = 2`

`Euler(`[*(int)Precision=53*]`)` → float :

Returns
`Real` The Euler–Mascheroni constant, exposed to python for testing of eigen numerical traits.

`IsComplex = 0`

`IsInteger = 0`

`IsSigned = 1`

`Log2(`[*(int)Precision=53*]`)` → float :

Returns
`Real` natural logarithm of 2, exposed to python for testing of eigen numerical traits.

`MulCost = 1`

`Pi(`[*(int)Precision=53*]`)` → float :

Returns
`Real` The constant, exposed to python for testing of eigen numerical traits.

`ReadCost = 1`

`Real`

alias of `float`

`RequireInitialization = 0`

`class Var`

The `Var` class is used to test to/from python converters for arbitrary precision `Real`

`property cpl`

one `Complex` variable to test reading from and writing to it.

`property val`

one `Real` variable for testing.

`abs`(*(complex)x*) → float :

Returns
the `Real` absolute value of the `Complex` argument. Depending on compilation options wraps `::boost::multiprecision::abs(…)` or std::abs(…) function.

abs((float)x) -> float :

return
the `Real` absolute value of the `Real` argument. Depending on compilation options wraps `::boost::multiprecision::abs(…)` or std::abs(…) function.

`acos`(*(complex)x*) → complex :

Returns
`Complex` the arc-cosine of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::acos(…)` or std::acos(…) function.

acos((float)x) -> float :

return
`Real` the arcus cosine of the argument. Depending on compilation options wraps `::boost::multiprecision::acos(…)` or std::acos(…) function.

`acosh`(*(complex)x*) → complex :

Returns
`Complex` the arc-hyperbolic cosine of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::acosh(…)` or std::acosh(…) function.

acosh((float)x) -> float :

return
`Real` the hyperbolic arcus cosine of the argument. Depending on compilation options wraps `::boost::multiprecision::acosh(…)` or std::acosh(…) function.

`arg`(*(complex)x*) → float :

Returns
`Real` the arg (Phase angle of complex in radians) of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::arg(…)` or std::arg(…) function.

`asin`(*(complex)x*) → complex :

Returns
`Complex` the arc-sine of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::asin(…)` or std::asin(…) function.

asin((float)x) -> float :

return
`Real` the arcus sine of the argument. Depending on compilation options wraps `::boost::multiprecision::asin(…)` or std::asin(…) function.

`asinh`(*(complex)x*) → complex :

Returns
`Complex` the arc-hyperbolic sine of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::asinh(…)` or std::asinh(…) function.

asinh((float)x) -> float :

return
`Real` the hyperbolic arcus sine of the argument. Depending on compilation options wraps `::boost::multiprecision::asinh(…)` or std::asinh(…) function.

`atan`(*(complex)x*) → complex :

Returns
`Complex` the arc-tangent of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::atan(…)` or std::atan(…) function.

atan((float)x) -> float :

return
`Real` the arcus tangent of the argument. Depending on compilation options wraps `::boost::multiprecision::atan(…)` or std::atan(…) function.

`atan2`(*(float)x*, *(float)y*) → float :

Returns
`Real` the arc tangent of y/x using the signs of the arguments `x` and `y` to determine the correct quadrant. Depending on compilation options wraps `::boost::multiprecision::atan2(…)` or std::atan2(…) function.

`atanh`(*(complex)x*) → complex :

Returns
`Complex` the arc-hyperbolic tangent of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::atanh(…)` or std::atanh(…) function.

atanh((float)x) -> float :

return
`Real` the hyperbolic arcus tangent of the argument. Depending on compilation options wraps `::boost::multiprecision::atanh(…)` or std::atanh(…) function.

`cbrt`(*(float)x*) → float :

Returns
`Real` cubic root of the argument. Depending on compilation options wraps `::boost::multiprecision::cbrt(…)` or std::cbrt(…) function.

`ceil`(*(float)x*) → float :

Returns
`Real` Computes the smallest integer value not less than arg. Depending on compilation options wraps `::boost::multiprecision::ceil(…)` or std::ceil(…) function.

`conj`(*(complex)x*) → complex :

Returns
the complex conjugation a `Complex` argument. Depending on compilation options wraps `::boost::multiprecision::conj(…)` or std::conj(…) function.

`cos`(*(complex)x*) → complex :

Returns
`Complex` the cosine of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::cos(…)` or std::cos(…) function.

cos((float)x) -> float :

return
`Real` the cosine of the `Real` argument in radians. Depending on compilation options wraps `::boost::multiprecision::cos(…)` or std::cos(…) function.

`cosh`(*(complex)x*) → complex :

Returns
`Complex` the hyperbolic cosine of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::cosh(…)` or std::cosh(…) function.

cosh((float)x) -> float :

return
`Real` the hyperbolic cosine of the `Real` argument in radians. Depending on compilation options wraps `::boost::multiprecision::cosh(…)` or std::cosh(…) function.

`cylBesselJ`(*(int)k*, *(float)x*) → float :

Returns
`Real` the Bessel Functions of the First Kind of the order `k` and the `Real` argument. See: <https://www.boost.org/doc/libs/1_77_0/libs/math/doc/html/math_toolkit/bessel/bessel_first.html>‘___

`defprec = 53`

`dummy_precision`() → float :

Returns
similar to the function `epsilon`, but assumes that last 10% of bits contain the numerical error only. This is sometimes used by Eigen when calling `isEqualFuzzy` to determine if values differ a lot or if they are vaguely close to each other.

`epsilon`(*[(int)Precision=53]*) → float :

Returns
`Real` returns the difference between `1.0` and the next representable value of the `Real` type. Wraps std::numeric_limits<Real>::epsilon() function.

epsilon((float)x) -> float :

return
`Real` returns the difference between `1.0` and the next representable value of the `Real` type. Wraps std::numeric_limits<Real>::epsilon() function.

`erf(`*(float)x*`)` → float :

Returns
`Real` Computes the error function of argument. Depending on compilation options wraps `::boost::multiprecision::erf(…)` or std::erf(…) function.

`erfc(`*(float)x*`)` → float :

Returns
`Real` Computes the complementary error function of argument, that is `1.0-erf(arg)`. Depending on compilation options wraps `::boost::multiprecision::erfc(…)` or std::erfc(…) function.

`exp(`*(complex)x*`)` → complex :

Returns
the base *e* exponential of a `Complex` argument. Depending on compilation options wraps `::boost::multiprecision::exp(…)` or std::exp(…) function.

exp((float)x) -> float :

return
the base *e* exponential of a `Real` argument. Depending on compilation options wraps `::boost::multiprecision::exp(…)` or std::exp(…) function.

`exp2(`*(float)x*`)` → float :

Returns
the base *2* exponential of a `Real` argument. Depending on compilation options wraps `::boost::multiprecision::exp2(…)` or std::exp2(…) function.

`expm1(`*(float)x*`)` → float :

Returns
the base *e* exponential of a `Real` argument minus `1.0`. Depending on compilation options wraps `::boost::multiprecision::expm1(…)` or std::expm1(…) function.

`fabs(`*(float)x*`)` → float :

Returns
the `Real` absolute value of the argument. Depending on compilation options wraps `::boost::multiprecision::abs(…)` or std::abs(…) function.

`factorial(`*(int)x*`)` → float :

Returns
`Real` the factorial of the `Real` argument. See: <https://www.boost.org/doc/libs/1_77_0/libs/math/doc/html/math_toolkit/factorials/sf_factorial.html>‘___

`floor(`*(float)x*`)` → float :

Returns
`Real` Computes the largest integer value not greater than arg. Depending on compilation options wraps `::boost::multiprecision::floor(…)` or std::floor(…) function.

`fma(`*(float)x*, *(float)y*, *(float)z*`)` → float :

Returns
`Real` - computes `(x*y) + z` as if to infinite precision and rounded only once to fit the result type. Depending on compilation options wraps `::boost::multiprecision::fma(…)` or std::fma(…) function.

`fmod`(*(float)x*, *(float)y*) → float :

Returns
`Real` the floating-point remainder of the division operation `x/y` of the arguments `x` and `y`. Depending on compilation options wraps `::boost::multiprecision::fmod(…)` or std::fmod(…) function.

`frexp`(*(float)x*) → tuple :

Returns
tuple of `(Real,int)`, decomposes given floating point `Real` argument into a normalized fraction and an integral power of two. Depending on compilation options wraps `::boost::multiprecision::frexp(…)` or std::frexp(…) function.

`fromBits`(*(str)bits*[, *(int)exp=0*[, *(int)sign=1*]]) → float :

Parameters

- `bits` – `str` - a string containing '0', '1' characters.
- `exp` – `int` - the binary exponent which shifts the bits.
- `sign` – `int` - the sign, should be -1 or +1, but it is not checked. It multiplies the result when construction from bits is finished.

Returns
`RealHP<N>` constructed from string containing '0', '1' bits. This is for debugging purposes, rather slow.

`getDecomposedReal`(*(float)x*) → dict :

Returns
`dict` - the dictionary with the debug information how the DecomposedReal class sees this type. This is for debugging purposes, rather slow. Includes result from fpclassify function call, a binary representation and other useful info. See also *fromBits*.

`getDemangledName`() → str :

Returns
`string` - the demangled C++ typnename of `RealHP<N>`.

`getDemangledNameComplex`() → str :

Returns
`string` - the demangled C++ typnename of `ComplexHP<N>`.

`getFloatDistanceULP`(*(float)arg1*, *(float)arg2*) → float :

Returns
an integer value stored in `RealHP<N>`, the ULP distance calculated by boost::math::float_distance, also see Floating-point Comparison and Prof. Kahan paper about this topic.

Warning

The returned value is the **directed distance** between two arguments, this means that it can be negative.

`getRawBits`(*(float)x*) → str :

Returns
`string` - the raw bits in memory representing this type. Be careful: it only checks the system endianness and either prints bytes in reverse order or not.

Does not make any attempts to further interpret the bits of: sign, exponent or significand (on a typical x86 processor they are printed in that order), and different processors might store them differently. It is not useful for types which internally use a pointer because for them this function prints not the floating point number but a pointer. This is for debugging purposes.

`hasInfinityNan = True`

`highest(`[*(int)Precision=53*]`)` → float :

Returns
`Real` returns the largest finite value of the `Real` type. Wraps std::numeric_limits<Real>::max() function.

`hypot(`*(float)x*, *(float)y*`)` → float :

Returns
`Real` the square root of the sum of the squares of `x` and `y`, without undue overflow or underflow at intermediate stages of the computation. Depending on compilation options wraps `::boost::multiprecision::hypot(…)` or std::hypot(…) function.

`ilogb(`*(float)x*`)` → float :

Returns
`Real` extracts the value of the unbiased exponent from the floating-point argument arg, and returns it as a signed integer value. Depending on compilation options wraps `::boost::multiprecision::ilogb(…)` or std::ilogb(…) function.

`imag(`*(complex)x*`)` → float :

Returns
the imag part of a `Complex` argument. Depending on compilation options wraps `::boost::multiprecision::imag(…)` or std::imag(…) function.

`isApprox(`*(float)a*, *(float)b*, *(float)eps*`)` → bool :

Returns
`bool`, True if `a` is approximately equal `b` with provided `eps`, see also here

`isApproxOrLessThan(`*(float)a*, *(float)b*, *(float)eps*`)` → bool :

Returns
`bool`, True if `a` is approximately less than or equal `b` with provided `eps`, see also here

`isEqualFuzzy(`*(float)arg1*, *(float)arg2*, *(float)arg3*`)` → bool :

Returns
`bool`, `True` if the absolute difference between two numbers is smaller than std::numeric_limits<Real>::epsilon()

`isMuchSmallerThan(`*(float)a*, *(float)b*, *(float)eps*`)` → bool :

Returns
`bool`, True if `a` is less than `b` with provided `eps`, see also here

`isfinite(`*(float)x*`)` → bool :

Returns
`bool` indicating if the `Real` argument is Inf. Depending on compilation options wraps `::boost::multiprecision::isfinite(…)` or std::isfinite(…) function.

`isinf(`*(float)x*`)` → bool :

Returns
`bool` indicating if the `Real` argument is Inf. Depending on compilation options wraps `::boost::multiprecision::isinf(…)` or std::isinf(…) function.

`isnan(`*(float)x*`)` → bool :

Returns
`bool` indicating if the `Real` argument is NaN. Depending on compilation options wraps `::boost::multiprecision::isnan(…)` or std::isnan(…) function.

`laguerre(`*(int)n*, *(int)m*, *(float)x*`)` → float :

Returns
`Real` the Laguerre polynomial of the orders `n`, `m` and the `Real` argument. See: <https://www.boost.org/doc/libs/1_77_0/libs/math/doc/html/math_toolkit/sf_poly/laguerre.html>‘___

`ldexp(`*(float)x*, *(int)y*`)` → float :

Returns
Multiplies a floating point value `x` by the number 2 raised to the `exp` power. Depending on compilation options wraps `::boost::multiprecision::ldexp(…)` or std::ldexp(…) function.

`lgamma(`*(float)x*`)` → float :

Returns
`Real` Computes the natural logarithm of the absolute value of the gamma function of arg. Depending on compilation options wraps `::boost::multiprecision::lgamma(…)` or std::lgamma(…) function.

`log(`*(complex)x*`)` → complex :

Returns
the `Complex` natural (base *e*) logarithm of a complex value z with a branch cut along the negative real axis. Depending on compilation options wraps `::boost::multiprecision::log(…)` or std::log(…) function.

log((float)x) -> float :

return
the `Real` natural (base *e*) logarithm of a real value. Depending on compilation options wraps `::boost::multiprecision::log(…)` or std::log(…) function.

`log10(`*(complex)x*`)` → complex :

Returns
the `Complex` (base *10*) logarithm of a complex value z with a branch cut along the negative real axis. Depending on compilation options wraps `::boost::multiprecision::log10(…)` or std::log10(…) function.

log10((float)x) -> float :

return
the `Real` decimal (base `10`) logarithm of a real value. Depending on compilation options wraps `::boost::multiprecision::log10(…)` or std::log10(…) function.

`log1p(`*(float)x*`)` → float :

Returns
the `Real` natural (base e) logarithm of `1+argument`. Depending on compilation options wraps `::boost::multiprecision::log1p(…)` or std::log1p(…) function.

`log2(`*(float)x*`)` → float :

Returns
the `Real` binary (base 2) logarithm of a real value. Depending on compilation options wraps `::boost::multiprecision::log2(…)` or std::log2(…) function.

`logb(`*(float)x*`)` → float :

Returns
Extracts the value of the unbiased radix-independent exponent from the floating-point argument arg, and returns it as a floating-point value. Depending on compilation options wraps `::boost::multiprecision::logb(…)` or std::logb(…) function.

`lowest(`$\left[$*(int)Precision=53*$\right]$`)` → float :

Returns
`Real` returns the lowest (negative) finite value of the `Real` type. Wraps std::numeric_limits<Real>::lowest() function.

`max(`*(float)x*, *(float)y*`)` → float :

Returns
`Real` larger of the two arguments. Depending on compilation options wraps `::boost::multiprecision::max(…)` or std::max(…) function.

`max_exp2 = 1024`

`min(`*(float)x*, *(float)y*`)` → float :

Returns
`Real` smaller of the two arguments. Depending on compilation options wraps `::boost::multiprecision::min(…)` or std::min(…) function.

`modf(`*(float)x*`)` → tuple :

Returns
tuple of `(Real,Real)`, decomposes given floating point `Real` into integral and fractional parts, each having the same type and sign as x. Depending on compilation options wraps `::boost::multiprecision::modf(…)` or std::modf(…) function.

`polar(`*(float)x*, *(float)y*`)` → complex :

Returns
`Complex` the polar (Complex from polar components) of the `Real` rho (length), `Real` theta (angle) arguments in radians. Depending on compilation options wraps `::boost::multiprecision::polar(…)` or std::polar(…) function.

`pow(`*(complex)x*, *(complex)pow*`)` → complex :

Returns
the `Complex` complex arg1 raised to the `Complex` power arg2. Depending on compilation options wraps `::boost::multiprecision::pow(…)` or std::pow(…) function.

pow((float)x, (float)y) -> float :

return
`Real` the value of `base` raised to the power `exp`. Depending on compilation options wraps `::boost::multiprecision::pow(…)` or std::pow(…) function.

`proj(`*(complex)x*`)` → complex :

Returns
`Complex` the proj (projection of the complex number onto the Riemann sphere) of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::proj(…)` or std::proj(…) function.

`random()` → float :

Returns
`Real` a symmetric random number in interval `(-1,1)`. Used by Eigen.

random((float)a, (float)b) -> float :

return
`Real` a random number in interval `(a,b)`. Used by Eigen.

`real(`*(complex)x*`)` → float :

Returns
the real part of a `Complex` argument. Depending on compilation options wraps `::boost::multiprecision::real(…)` or std::real(…) function.

`remainder(`*(float)x*, *(float)y*`)` → float :

Returns
`Real` the IEEE remainder of the floating point division operation `x/y`. Depending on compilation options wraps `::boost::multiprecision::remainder(…)` or std::remainder(…) function.

`remquo(`*(float)x*, *(float)y*`)` → tuple :

Returns
tuple of `(Real,long)`, the floating-point remainder of the division operation `x/y` as the std::remainder() function does. Additionally, the sign and at least the three of the last bits of `x/y` are returned, sufficient to determine the octant of the result within a period. Depending on compilation options wraps `::boost::multiprecision::remquo(…)` or std::remquo(…) function.

`rint(`*(float)x*`)` → float :

Returns
Rounds the floating-point argument arg to an integer value (in floating-point format), using the current rounding mode. Depending on compilation options wraps `::boost::multiprecision::rint(…)` or std::rint(…) function.

`round(`*(float)x*`)` → float :

Returns
`Real` the nearest integer value to arg (in floating-point format), rounding halfway cases away from zero, regardless of the current rounding mode.. Depending on compilation options wraps `::boost::multiprecision::round(…)` or std::round(…) function.

`roundTrip(`*(float)x*`)` → float :

Returns
`Real` returns the argument `x`. Can be used to convert type to native RealHP<N> accuracy.

`sgn(`*(float)x*`)` → int :

Returns
`int` the sign of the argument: `-1`, `0` or `1`.

`sign`(*(float)x*) → int :

Returns
`int` the sign of the argument: `-1`, `0` or `1`.

`sin`(*(complex)x*) → complex :

Returns
`Complex` the sine of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::sin(…)` or std::sin(…) function.

sin((float)x) -> float :

return
`Real` the sine of the `Real` argument in radians. Depending on compilation options wraps `::boost::multiprecision::sin(…)` or std::sin(…) function.

`sinh`(*(complex)x*) → complex :

Returns
`Complex` the hyperbolic sine of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::sinh(…)` or std::sinh(…) function.

sinh((float)x) -> float :

return
`Real` the hyperbolic sine of the `Real` argument in radians. Depending on compilation options wraps `::boost::multiprecision::sinh(…)` or std::sinh(…) function.

`smallest_positive`() → float :

Returns
`Real` the smallest number greater than zero. Wraps std::numeric_limits<Real>::min()

`sphericalHarmonic`(*(int)l*, *(int)m*, *(float)theta*, *(float)phi*) → complex :

Returns
`Real` the spherical harmonic polynomial of the orders `l` (unsigned int), `m` (signed int) and the `Real` arguments `theta` and `phi`. See: <https://www.boost.org/doc/libs/1_77_0/libs/math/doc/html/math_toolkit/sf_poly/sph_harm.html>‘___

`sqrt`(*(complex)x*) → complex :

Returns
the `Complex` square root of `Complex` argument. Depending on compilation options wraps `::boost::multiprecision::sqrt(…)` or std::sqrt(…) function.

sqrt((float)x) -> float :

return
`Real` square root of the argument. Depending on compilation options wraps `::boost::multiprecision::sqrt(…)` or std::sqrt(…) function.

`squaredNorm`(*(complex)x*) → float :

Returns
`Real` the norm (squared magnitude) of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::norm(…)` or std::norm(…) function.

`tan`(*(complex)x*) → complex :

Returns
`Complex` the tangent of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::tan(…)` or std::tan(…) function.

tan((float)x) -> float :

return
`Real` the tangent of the `Real` argument in radians. Depending on compilation options wraps `::boost::multiprecision::tan(…)` or std::tan(…) function.

`tanh`(*(complex)x*) → complex :

Returns
`Complex` the hyperbolic tangent of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::tanh(…)` or std::tanh(…) function.

tanh((float)x) -> float :

return
`Real` the hyperbolic tangent of the `Real` argument in radians. Depending on compilation options wraps `::boost::multiprecision::tanh(…)` or std::tanh(…) function.

`testArray`() → None :

This function tests call to `std::vector::data(…)` function in order to extract the array.

`testCgalNumTraits = True`

`testConstants`() → None :

This function tests lib/high-precision/Constants.hpp, the yade::math::ConstantsHP<N>, former yade::Mathr constants.

`tgamma`(*(float)x*) → float :

Returns
`Real` Computes the gamma function of arg. Depending on compilation options wraps `::boost::multiprecision::tgamma(…)` or std::tgamma(…) function.

`toDouble`(*(float)x*) → float :

Returns
`float` converts `Real` type to `double` and returns a native python `float`.

`toHP1`(*(float)x*) → float :

Returns
`RealHP<1>` converted from argument `RealHP<1>` as a result of `static_cast<RealHP<1>>(arg)`.

`toHP2`(*(float)x*) → yade._minieigenHP.HP2.Real :

Returns
`RealHP<2>` converted from argument `RealHP<1>` as a result of `static_cast<RealHP<2>>(arg)`.

`toInt`(*(float)x*) → int :

Returns
`int` converts `Real` type to `int` and returns a native python `int`.

`toLong`(*(float)x*) → int :

Returns
`int` converts `Real` type to `long int` and returns a native python `int`.

`toLongDouble`(*(float)x*) → float :

Returns
`float` converts `Real` type to `long double` and returns a native python `float`.

`trunc`(*(float)x*) → float :

Returns
`Real` the nearest integer not greater in magnitude than arg. Depending on compilation options wraps `::boost::multiprecision::trunc(…)` or std::trunc(…) function.

`class yade._math.HP2`

`AddCost = 2`

`CGAL_Is_finite`(*(yade._minieigenHP.HP2.Real)x*) → bool :

CGAL's function `Is_finite`, as described in CGAL algebraic foundations exposed to python for testing of CGAL numerical traits.

Returns
`bool` indicating if the `Real` argument is finite.

`CGAL_Is_valid`(*(yade._minieigenHP.HP2.Real)x*) → bool :

CGAL's function `Is_valid`, as described in CGAL algebraic foundations exposed to python for testing of CGAL numerical traits.

Returns
`bool` indicating if the `Real` argument is valid. Checks are performed against NaN and Inf.

`CGAL_Kth_root`(*(int)arg1*, *(yade._minieigenHP.HP2.Real)x*) → yade._minieigenHP.HP2.Real :

CGAL's function `Kth_root`, as described in CGAL algebraic foundations exposed to python for testing of CGAL numerical traits.

Returns
`Real` the k-th root of argument.

`CGAL_Sgn`(*(yade._minieigenHP.HP2.Real)x*) → int :

CGAL's function `Sgn`, as described in CGAL algebraic foundations exposed to python for testing of CGAL numerical traits.

Returns
sign of the argument, can be `-1`, `0` or `1`. Not very useful in python. In C++ it is useful to obtain a sign of an expression with exact accuracy, CGAL starts using MPFR internally for this when the approximate interval contains zero inside it.

`CGAL_Sqrt`(*(yade._minieigenHP.HP2.Real)x*) → yade._minieigenHP.HP2.Real :

CGAL's function `Sqrt`, as described in CGAL algebraic foundations exposed to python for testing of CGAL numerical traits.

Returns
`Real` the square root of argument.

`CGAL_Square`(*(yade._minieigenHP.HP2.Real)x*) → yade._minieigenHP.HP2.Real :

CGAL's function `Square`, as described in CGAL algebraic foundations exposed to python for testing of CGAL numerical traits.

Returns
`Real` the argument squared.

`CGAL_To_interval`(*(yade._minieigenHP.HP2.Real)x*) → tuple :

CGAL's function `To_interval`, as described in CGAL algebraic foundations exposed to python for testing of CGAL numerical traits.

Returns
`(double,double)` tuple inside which the high-precision `Real` argument resides.

`CGAL_simpleTest`() → yade._minieigenHP.HP2.Real :

Tests a simple CGAL calculation. Distance between plane and point, uses CGAL's sqrt and pow.

Returns
3.0

`Catalan`([*(int)Precision=113*]) → yade._minieigenHP.HP2.Real :

Returns
`Real` The catalan constant, exposed to python for testing of eigen numerical traits.

`class Complex`

The Complex type.

`__init__`(*(object)arg1*) → None

___init___((object)arg1, (object)obj) -> object

___init___((object)arg1, (complex)z) -> object

___init___((object)arg1, (float)d) -> object

___init___((object)arg1, (int)i) -> object

___init___((object)arg1, (str)str) -> object

___init___((object)arg1, (object)re, (object)im) -> object

___init___((object)arg1, (float)a, (float)b) -> object

___init___((object)arg1, (int)i, (int)j) -> object

___init___((object)arg1, (str)str1, (str)str2) -> object

`property imag`

None((yade._minieigenHP.HP2.Complex)arg1) -> object

`levelComplexHPMethod`(*(HP2.Complex)arg1*) → int

`property levelHP`

None((yade._minieigenHP.HP2.Complex)arg1) -> int

`property real`

None((yade._minieigenHP.HP2.Complex)arg1) -> object

`ComplexAddCost = 4`

`ComplexMulCost = 12`

`ComplexReadCost = 2`

`Euler`([*(int)Precision=113*]) → yade._minieigenHP.HP2.Real :

Returns
`Real` The Euler–Mascheroni constant, exposed to python for testing of eigen numerical traits.

`IsComplex = 0`

`IsInteger = 0`

`IsSigned = 1`

`Log2(`[*(int)Precision=113*]`)` → yade._minieigenHP.HP2.Real :

Returns
`Real` natural logarithm of 2, exposed to python for testing of eigen numerical traits.

`MulCost = 2`

`Pi(`[*(int)Precision=113*]`)` → yade._minieigenHP.HP2.Real :

Returns
`Real` The constant, exposed to python for testing of eigen numerical traits.

`ReadCost = 1`

`class Real`

The Real type.

`__init__(`*(object)arg1*`)` → None

___init___((object)arg1, (object)obj) -> object

___init___((object)arg1, (float)d) -> object

___init___((object)arg1, (int)i) -> object

___init___((object)arg1, (str)str) -> object

`property imag`

None((yade._minieigenHP.HP2.Real)arg1) -> yade._minieigenHP.HP2.Real

`property levelHP`

None((yade._minieigenHP.HP2.Real)arg1) -> int

`levelRealHPMethod(`*(HP2.Real)arg1*`)` → int

`property real`

None((yade._minieigenHP.HP2.Real)arg1) -> yade._minieigenHP.HP2.Real

`sqrt(`*(HP2.Real)arg1*`)` → *HP2.Real*

`RequireInitialization = 1`

`class Var`

The `Var` class is used to test to/from python converters for arbitrary precision `Real`

`property cpl`

one `Complex` variable to test reading from and writing to it.

`property val`

one `Real` variable for testing.

`abs(`*(yade._minieigenHP.HP2.Complex)x*`)` → yade._minieigenHP.HP2.Real :

Returns
the `Real` absolute value of the `Complex` argument. Depending on compilation options wraps `::boost::multiprecision::abs(…)` or std::abs(…) function.

abs((yade._minieigenHP.HP2.Real)x) -> yade._minieigenHP.HP2.Real :

return
the `Real` absolute value of the `Real` argument. Depending on compilation options wraps `::boost::multiprecision::abs(…)` or std::abs(…) function.

`acos`(*(yade._minieigenHP.HP2.Complex)x*) → yade._minieigenHP.HP2.Complex :

Returns
`Complex` the arc-cosine of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::acos(…)` or std::acos(…) function.

acos((yade._minieigenHP.HP2.Real)x) -> yade._minieigenHP.HP2.Real :

return
`Real` the arcus cosine of the argument. Depending on compilation options wraps `::boost::multiprecision::acos(…)` or std::acos(…) function.

`acosh`(*(yade._minieigenHP.HP2.Complex)x*) → yade._minieigenHP.HP2.Complex :

Returns
`Complex` the arc-hyperbolic cosine of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::acosh(…)` or std::acosh(…) function.

acosh((yade._minieigenHP.HP2.Real)x) -> yade._minieigenHP.HP2.Real :

return
`Real` the hyperbolic arcus cosine of the argument. Depending on compilation options wraps `::boost::multiprecision::acosh(…)` or std::acosh(…) function.

`arg`(*(yade._minieigenHP.HP2.Complex)x*) → yade._minieigenHP.HP2.Real :

Returns
`Real` the arg (Phase angle of complex in radians) of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::arg(…)` or std::arg(…) function.

`asin`(*(yade._minieigenHP.HP2.Complex)x*) → yade._minieigenHP.HP2.Complex :

Returns
`Complex` the arc-sine of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::asin(…)` or std::asin(…) function.

asin((yade._minieigenHP.HP2.Real)x) -> yade._minieigenHP.HP2.Real :

return
`Real` the arcus sine of the argument. Depending on compilation options wraps `::boost::multiprecision::asin(…)` or std::asin(…) function.

`asinh`(*(yade._minieigenHP.HP2.Complex)x*) → yade._minieigenHP.HP2.Complex :

Returns
`Complex` the arc-hyperbolic sine of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::asinh(…)` or std::asinh(…) function.

asinh((yade._minieigenHP.HP2.Real)x) -> yade._minieigenHP.HP2.Real :

return
`Real` the hyperbolic arcus sine of the argument. Depending on compilation options wraps `::boost::multiprecision::asinh(…)` or std::asinh(…) function.

`atan`(*(yade._minieigenHP.HP2.Complex)x*) → yade._minieigenHP.HP2.Complex :

Returns
`Complex` the arc-tangent of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::atan(…)` or std::atan(…) function.

atan((yade._minieigenHP.HP2.Real)x) -> yade._minieigenHP.HP2.Real :

return
`Real` the arcus tangent of the argument. Depending on compilation options wraps `::boost::multiprecision::atan(…)` or std::atan(…) function.

`atan2`(*(yade._minieigenHP.HP2.Real)x*, *(yade._minieigenHP.HP2.Real)y*) → yade._minieigenHP.HP2.Real :

Returns
`Real` the arc tangent of y/x using the signs of the arguments `x` and `y` to determine the correct quadrant. Depending on compilation options wraps `::boost::multiprecision::atan2(…)` or std::atan2(…) function.

`atanh`(*(yade._minieigenHP.HP2.Complex)x*) → yade._minieigenHP.HP2.Complex :

Returns
`Complex` the arc-hyperbolic tangent of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::atanh(…)` or std::atanh(…) function.

atanh((yade._minieigenHP.HP2.Real)x) -> yade._minieigenHP.HP2.Real :

return
`Real` the hyperbolic arcus tangent of the argument. Depending on compilation options wraps `::boost::multiprecision::atanh(…)` or std::atanh(…) function.

`cbrt`(*(yade._minieigenHP.HP2.Real)x*) → yade._minieigenHP.HP2.Real :

Returns
`Real` cubic root of the argument. Depending on compilation options wraps `::boost::multiprecision::cbrt(…)` or std::cbrt(…) function.

`ceil`(*(yade._minieigenHP.HP2.Real)x*) → yade._minieigenHP.HP2.Real :

Returns
`Real` Computes the smallest integer value not less than arg. Depending on compilation options wraps `::boost::multiprecision::ceil(…)` or std::ceil(…) function.

`conj`(*(yade._minieigenHP.HP2.Complex)x*) → yade._minieigenHP.HP2.Complex :

Returns
the complex conjugation a `Complex` argument. Depending on compilation options wraps `::boost::multiprecision::conj(…)` or std::conj(…) function.

`cos`(*(yade._minieigenHP.HP2.Complex)x*) → yade._minieigenHP.HP2.Complex :

Returns
`Complex` the cosine of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::cos(…)` or std::cos(…) function.

cos((yade._minieigenHP.HP2.Real)x) -> yade._minieigenHP.HP2.Real :

return
`Real` the cosine of the `Real` argument in radians. Depending on compilation options wraps `::boost::multiprecision::cos(…)` or std::cos(…) function.

`cosh(`*(yade._minieigenHP.HP2.Complex)x*`)` → yade._minieigenHP.HP2.Complex :

Returns
`Complex` the hyperbolic cosine of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::cosh(…)` or std::cosh(…) function.

cosh((yade._minieigenHP.HP2.Real)x) -> yade._minieigenHP.HP2.Real :

return
`Real` the hyperbolic cosine of the `Real` argument in radians. Depending on compilation options wraps `::boost::multiprecision::cosh(…)` or std::cosh(…) function.

`cylBesselJ(`*(int)k*, *(yade._minieigenHP.HP2.Real)x*`)` → yade._minieigenHP.HP2.Real :

Returns
`Real` the Bessel Functions of the First Kind of the order `k` and the `Real` argument. See: <https://www.boost.org/doc/libs/1_77_0/libs/math/doc/html/math_toolkit/bessel/bessel_first.html>‘___

`defprec = 113`

`dummy_precision()` → yade._minieigenHP.HP2.Real :

Returns
similar to the function `epsilon`, but assumes that last 10% of bits contain the numerical error only. This is sometimes used by Eigen when calling `isEqualFuzzy` to determine if values differ a lot or if they are vaguely close to each other.

`epsilon(`[*(int)Precision=113*]`)` → yade._minieigenHP.HP2.Real :

Returns
`Real` returns the difference between `1.0` and the next representable value of the `Real` type. Wraps std::numeric_limits<Real>::epsilon() function.

epsilon((yade._minieigenHP.HP2.Real)x) -> yade._minieigenHP.HP2.Real :

return
`Real` returns the difference between `1.0` and the next representable value of the `Real` type. Wraps std::numeric_limits<Real>::epsilon() function.

`erf(`*(yade._minieigenHP.HP2.Real)x*`)` → yade._minieigenHP.HP2.Real :

Returns
`Real` Computes the error function of argument. Depending on compilation options wraps `::boost::multiprecision::erf(…)` or std::erf(…) function.

`erfc(`*(yade._minieigenHP.HP2.Real)x*`)` → yade._minieigenHP.HP2.Real :

Returns
`Real` Computes the complementary error function of argument, that is `1.0-erf(arg)`. Depending on compilation options wraps `::boost::multiprecision::erfc(…)` or std::erfc(…) function.

`exp`(*(yade._minieigenHP.HP2.Complex)x*) → yade._minieigenHP.HP2.Complex :

Returns
the base *e* exponential of a `Complex` argument. Depending on compilation options wraps `::boost::multiprecision::exp(…)` or std::exp(…) function.

exp((yade._minieigenHP.HP2.Real)x) -> yade._minieigenHP.HP2.Real :

return
the base *e* exponential of a `Real` argument. Depending on compilation options wraps `::boost::multiprecision::exp(…)` or std::exp(…) function.

`exp2`(*(yade._minieigenHP.HP2.Real)x*) → yade._minieigenHP.HP2.Real :

Returns
the base *2* exponential of a `Real` argument. Depending on compilation options wraps `::boost::multiprecision::exp2(…)` or std::exp2(…) function.

`expm1`(*(yade._minieigenHP.HP2.Real)x*) → yade._minieigenHP.HP2.Real :

Returns
the base *e* exponential of a `Real` argument minus `1.0`. Depending on compilation options wraps `::boost::multiprecision::expm1(…)` or std::expm1(…) function.

`fabs`(*(yade._minieigenHP.HP2.Real)x*) → yade._minieigenHP.HP2.Real :

Returns
the `Real` absolute value of the argument. Depending on compilation options wraps `::boost::multiprecision::abs(…)` or std::abs(…) function.

`factorial`(*(int)x*) → yade._minieigenHP.HP2.Real :

Returns
`Real` the factorial of the `Real` argument. See: <https://www.boost.org/doc/libs/1_77_0/libs/math/doc/html/math_toolkit/factorials/sf_factorial.html>‘___

`floor`(*(yade._minieigenHP.HP2.Real)x*) → yade._minieigenHP.HP2.Real :

Returns
`Real` Computes the largest integer value not greater than arg. Depending on compilation options wraps `::boost::multiprecision::floor(…)` or std::floor(…) function.

`fma`(*(yade._minieigenHP.HP2.Real)x*, *(yade._minieigenHP.HP2.Real)y*, *(yade._minieigenHP.HP2.Real)z*) → yade._minieigenHP.HP2.Real :

Returns
`Real` - computes `(x*y) + z` as if to infinite precision and rounded only once to fit the result type. Depending on compilation options wraps `::boost::multiprecision::fma(…)` or std::fma(…) function.

`fmod`(*(yade._minieigenHP.HP2.Real)x*, *(yade._minieigenHP.HP2.Real)y*) → yade._minieigenHP.HP2.Real :

Returns
`Real` the floating-point remainder of the division operation `x/y` of the arguments `x` and `y`. Depending on compilation options wraps `::boost::multiprecision::fmod(…)` or std::fmod(…) function.

`frexp`(*(yade._minieigenHP.HP2.Real)x*) → tuple :

Returns
tuple of `(Real,int)`, decomposes given floating point `Real` argument into a normalized fraction and an integral power of two. Depending on compilation options wraps `::boost::multiprecision::frexp(…)` or std::frexp(…) function.

`fromBits`(*(str)bits*[, *(int)exp=0*[, *(int)sign=1*]]) → yade._minieigenHP.HP2.Real :

Parameters

- **bits** – `str` - a string containing '0', '1' characters.
- **exp** – `int` - the binary exponent which shifts the bits.
- **sign** – `int` - the sign, should be -1 or +1, but it is not checked. It multiplies the result when construction from bits is finished.

Returns

`RealHP<N>` constructed from string containing '0', '1' bits. This is for debugging purposes, rather slow.

`getDecomposedReal`(*(yade._minieigenHP.HP2.Real)x*) → dict :

Returns

`dict` - the dictionary with the debug information how the DecomposedReal class sees this type. This is for debugging purposes, rather slow. Includes result from fpclassify function call, a binary representation and other useful info. See also *fromBits*.

`getDemangledName`() → str :

Returns

`string` - the demangled C++ typnename of `RealHP<N>`.

`getDemangledNameComplex`() → str :

Returns

`string` - the demangled C++ typnename of `ComplexHP<N>`.

`getFloatDistanceULP`(*(yade._minieigenHP.HP2.Real)arg1*, *(yade._minieigenHP.HP2.Real)arg2*) → yade._minieigenHP.HP2.Real :

Returns

an integer value stored in `RealHP<N>`, the ULP distance calculated by boost::math::float_distance, also see Floating-point Comparison and Prof. Kahan paper about this topic.

Warning

The returned value is the **directed distance** between two arguments, this means that it can be negative.

`getRawBits`(*(yade._minieigenHP.HP2.Real)x*) → str :

Returns

`string` - the raw bits in memory representing this type. Be careful: it only checks the system endianness and either prints bytes in reverse order or not. Does not make any attempts to further interpret the bits of: sign, exponent or significand (on a typical x86 processor they are printed in that order), and different processors might store them differently. It is not useful for types which internally use a pointer because for them this function prints not the floating point number but a pointer. This is for debugging purposes.

`hasInfinityNan = True`

`highest`([*(int)Precision=113*]) → yade._minieigenHP.HP2.Real :

Returns

`Real` returns the largest finite value of the `Real` type. Wraps std::numeric_limits<Real>::max() function.

hypot(*(yade._minieigenHP.HP2.Real)x*, *(yade._minieigenHP.HP2.Real)y*) → yade._minieigenHP.HP2.Real :

Returns
`Real` the square root of the sum of the squares of `x` and `y`, without undue overflow or underflow at intermediate stages of the computation. Depending on compilation options wraps `::boost::multiprecision::hypot(…)` or std::hypot(…) function.

ilogb(*(yade._minieigenHP.HP2.Real)x*) → yade._minieigenHP.HP2.Real :

Returns
`Real` extracts the value of the unbiased exponent from the floating-point argument arg, and returns it as a signed integer value. Depending on compilation options wraps `::boost::multiprecision::ilogb(…)` or std::ilogb(…) function.

imag(*(yade._minieigenHP.HP2.Complex)x*) → yade._minieigenHP.HP2.Real :

Returns
the imag part of a `Complex` argument. Depending on compilation options wraps `::boost::multiprecision::imag(…)` or std::imag(…) function.

isApprox(*(yade._minieigenHP.HP2.Real)a*, *(yade._minieigenHP.HP2.Real)b*, *(yade._minieigenHP.HP2.Real)eps*) → bool :

Returns
`bool`, True if `a` is approximately equal `b` with provided `eps`, see also here

isApproxOrLessThan(*(yade._minieigenHP.HP2.Real)a*, *(yade._minieigenHP.HP2.Real)b*, *(yade._minieigenHP.HP2.Real)eps*) → bool :

Returns
`bool`, True if `a` is approximately less than or equal `b` with provided `eps`, see also here

isEqualFuzzy(*(yade._minieigenHP.HP2.Real)arg1*, *(yade._minieigenHP.HP2.Real)arg2*, *(yade._minieigenHP.HP2.Real)arg3*) → bool :

Returns
`bool`, `True` if the absolute difference between two numbers is smaller than std::numeric_limits<Real>::epsilon()

isMuchSmallerThan(*(yade._minieigenHP.HP2.Real)a*, *(yade._minieigenHP.HP2.Real)b*, *(yade._minieigenHP.HP2.Real)eps*) → bool :

Returns
`bool`, True if `a` is less than `b` with provided `eps`, see also here

isfinite(*(yade._minieigenHP.HP2.Real)x*) → bool :

Returns
`bool` indicating if the `Real` argument is Inf. Depending on compilation options wraps `::boost::multiprecision::isfinite(…)` or std::isfinite(…) function.

isinf(*(yade._minieigenHP.HP2.Real)x*) → bool :

Returns
`bool` indicating if the `Real` argument is Inf. Depending on compilation options wraps `::boost::multiprecision::isinf(…)` or std::isinf(…) function.

isnan(*(yade._minieigenHP.HP2.Real)x*) → bool :

Returns
`bool` indicating if the `Real` argument is NaN. Depending on compilation options wraps `::boost::multiprecision::isnan(…)` or std::isnan(…) function.

`laguerre`(*(int)n*, *(int)m*, *(yade._minieigenHP.HP2.Real)x*) → yade._minieigenHP.HP2.Real :

Returns
`Real` the Laguerre polynomial of the orders `n`, `m` and the `Real` argument. See: <https://www.boost.org/doc/libs/1_77_0/libs/math/doc/html/math_toolkit/sf_poly/laguerre.html>‘___

`ldexp`(*(yade._minieigenHP.HP2.Real)x*, *(int)y*) → yade._minieigenHP.HP2.Real :

Returns
Multiplies a floating point value `x` by the number 2 raised to the `exp` power. Depending on compilation options wraps `::boost::multiprecision::ldexp(…)` or std::ldexp(…) function.

`lgamma`(*(yade._minieigenHP.HP2.Real)x*) → yade._minieigenHP.HP2.Real :

Returns
`Real` Computes the natural logarithm of the absolute value of the gamma function of arg. Depending on compilation options wraps `::boost::multiprecision::lgamma(…)` or std::lgamma(…) function.

`log`(*(yade._minieigenHP.HP2.Complex)x*) → yade._minieigenHP.HP2.Complex :

Returns
the `Complex` natural (base *e*) logarithm of a complex value z with a branch cut along the negative real axis. Depending on compilation options wraps `::boost::multiprecision::log(…)` or std::log(…) function.

log((yade._minieigenHP.HP2.Real)x) -> yade._minieigenHP.HP2.Real :

return
the `Real` natural (base *e*) logarithm of a real value. Depending on compilation options wraps `::boost::multiprecision::log(…)` or std::log(…) function.

`log10`(*(yade._minieigenHP.HP2.Complex)x*) → yade._minieigenHP.HP2.Complex :

Returns
the `Complex` (base *10*) logarithm of a complex value z with a branch cut along the negative real axis. Depending on compilation options wraps `::boost::multiprecision::log10(…)` or std::log10(…) function.

log10((yade._minieigenHP.HP2.Real)x) -> yade._minieigenHP.HP2.Real :

return
the `Real` decimal (base `10`) logarithm of a real value. Depending on compilation options wraps `::boost::multiprecision::log10(…)` or std::log10(…) function.

`log1p`(*(yade._minieigenHP.HP2.Real)x*) → yade._minieigenHP.HP2.Real :

Returns
the `Real` natural (base *e*) logarithm of `1+argument`. Depending on compilation options wraps `::boost::multiprecision::log1p(…)` or std::log1p(…) function.

`log2`(*(yade._minieigenHP.HP2.Real)x*) → yade._minieigenHP.HP2.Real :

Returns
the `Real` binary (base `2`) logarithm of a real value. Depending on compilation options wraps `::boost::multiprecision::log2(…)` or std::log2(…) function.

`logb`(*(yade._minieigenHP.HP2.Real)x*) → yade._minieigenHP.HP2.Real :

Returns
Extracts the value of the unbiased radix-independent exponent from the floating-point argument arg, and returns it as a floating-point value. Depending on compilation options wraps `::boost::multiprecision::logb(…)` or std::logb(…) function.

`lowest`([*(int)Precision=113*]) → yade._minieigenHP.HP2.Real :

Returns
`Real` returns the lowest (negative) finite value of the `Real` type. Wraps std::numeric_limits<Real>::lowest() function.

`max`(*(yade._minieigenHP.HP2.Real)x*, *(yade._minieigenHP.HP2.Real)y*) → yade._minieigenHP.HP2.Real :

Returns
`Real` larger of the two arguments. Depending on compilation options wraps `::boost::multiprecision::max(…)` or std::max(…) function.

`max_exp2 = 16384`

`min`(*(yade._minieigenHP.HP2.Real)x*, *(yade._minieigenHP.HP2.Real)y*) → yade._minieigenHP.HP2.Real :

Returns
`Real` smaller of the two arguments. Depending on compilation options wraps `::boost::multiprecision::min(…)` or std::min(…) function.

`modf`(*(yade._minieigenHP.HP2.Real)x*) → tuple :

Returns
tuple of `(Real,Real)`, decomposes given floating point `Real` into integral and fractional parts, each having the same type and sign as x. Depending on compilation options wraps `::boost::multiprecision::modf(…)` or std::modf(…) function.

`polar`(*(yade._minieigenHP.HP2.Real)x*, *(yade._minieigenHP.HP2.Real)y*) → yade._minieigenHP.HP2.Complex :

Returns
`Complex` the polar (Complex from polar components) of the `Real` rho (length), `Real` theta (angle) arguments in radians. Depending on compilation options wraps `::boost::multiprecision::polar(…)` or std::polar(…) function.

`pow`(*(yade._minieigenHP.HP2.Complex)x*, *(yade._minieigenHP.HP2.Complex)pow*) → yade._minieigenHP.HP2.Complex :

Returns
the `Complex` complex arg1 raised to the `Complex` power arg2. Depending on compilation options wraps `::boost::multiprecision::pow(…)` or std::pow(…) function.

pow((yade._minieigenHP.HP2.Real)x, (yade._minieigenHP.HP2.Real)y) -> yade._minieigenHP.HP2.Real :

return
`Real` the value of `base` raised to the power `exp`. Depending on compilation options wraps `::boost::multiprecision::pow(…)` or std::pow(…) function.

`proj`(*(yade._minieigenHP.HP2.Complex)x*) → yade._minieigenHP.HP2.Complex :

Returns
`Complex` the proj (projection of the complex number onto the Riemann sphere) of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::proj(…)` or std::proj(…) function.

`random()` → yade._minieigenHP.HP2.Real :

Returns
`Real` a symmetric random number in interval `(-1,1)`. Used by Eigen.

random((yade._minieigenHP.HP2.Real)a, (yade._minieigenHP.HP2.Real)b) -> yade._minieigenHP.HP2.Real :

return
`Real` a random number in interval `(a,b)`. Used by Eigen.

`real`(*(yade._minieigenHP.HP2.Complex)x*) → yade._minieigenHP.HP2.Real :

Returns
the real part of a `Complex` argument. Depending on compilation options wraps `::boost::multiprecision::real(…)` or std::real(…) function.

`remainder`(*(yade._minieigenHP.HP2.Real)x*, *(yade._minieigenHP.HP2.Real)y*) → yade._minieigenHP.HP2.Real :

Returns
`Real` the IEEE remainder of the floating point division operation `x/y`. Depending on compilation options wraps `::boost::multiprecision::remainder(…)` or std::remainder(…) function.

`remquo`(*(yade._minieigenHP.HP2.Real)x*, *(yade._minieigenHP.HP2.Real)y*) → tuple :

Returns
tuple of `(Real,long)`, the floating-point remainder of the division operation `x/y` as the std::remainder() function does. Additionally, the sign and at least the three of the last bits of `x/y` are returned, sufficient to determine the octant of the result within a period. Depending on compilation options wraps `::boost::multiprecision::remquo(…)` or std::remquo(…) function.

`rint`(*(yade._minieigenHP.HP2.Real)x*) → yade._minieigenHP.HP2.Real :

Returns
Rounds the floating-point argument arg to an integer value (in floating-point format), using the current rounding mode. Depending on compilation options wraps `::boost::multiprecision::rint(…)` or std::rint(…) function.

`round`(*(yade._minieigenHP.HP2.Real)x*) → yade._minieigenHP.HP2.Real :

Returns
`Real` the nearest integer value to arg (in floating-point format), rounding halfway cases away from zero, regardless of the current rounding mode.. Depending on compilation options wraps `::boost::multiprecision::round(…)` or std::round(…) function.

`roundTrip`(*(yade._minieigenHP.HP2.Real)x*) → yade._minieigenHP.HP2.Real :

Returns
`Real` returns the argument `x`. Can be used to convert type to native RealHP<N> accuracy.

`sgn`(*(yade._minieigenHP.HP2.Real)x*) → int :

Returns
`int` the sign of the argument: `-1`, `0` or `1`.

`sign`(*(yade._minieigenHP.HP2.Real)x*) → int :

Returns
`int` the sign of the argument: `-1`, `0` or `1`.

`sin`(*(yade._minieigenHP.HP2.Complex)x*) → yade._minieigenHP.HP2.Complex :

Returns
`Complex` the sine of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::sin(…)` or std::sin(…) function.

sin((yade._minieigenHP.HP2.Real)x) -> yade._minieigenHP.HP2.Real :

return
`Real` the sine of the `Real` argument in radians. Depending on compilation options wraps `::boost::multiprecision::sin(…)` or std::sin(…) function.

`sinh`(*(yade._minieigenHP.HP2.Complex)x*) → yade._minieigenHP.HP2.Complex :

Returns
`Complex` the hyperbolic sine of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::sinh(…)` or std::sinh(…) function.

sinh((yade._minieigenHP.HP2.Real)x) -> yade._minieigenHP.HP2.Real :

return
`Real` the hyperbolic sine of the `Real` argument in radians. Depending on compilation options wraps `::boost::multiprecision::sinh(…)` or std::sinh(…) function.

`smallest_positive`() → yade._minieigenHP.HP2.Real :

Returns
`Real` the smallest number greater than zero. Wraps std::numeric_limits<Real>::min()

`sphericalHarmonic`(*(int)l*, *(int)m*, *(yade._minieigenHP.HP2.Real)theta*, *(yade._minieigenHP.HP2.Real)phi*) → yade._minieigenHP.HP2.Complex :

Returns
`Real` the spherical harmonic polynomial of the orders `l` (unsigned int), `m` (signed int) and the `Real` arguments `theta` and `phi`. See: <https://www.boost.org/doc/libs/1_77_0/libs/math/doc/html/math_toolkit/sf_poly/sph_harm.html>‘___

`sqrt`(*(yade._minieigenHP.HP2.Complex)x*) → yade._minieigenHP.HP2.Complex :

Returns
the `Complex` square root of `Complex` argument. Depending on compilation options wraps `::boost::multiprecision::sqrt(…)` or std::sqrt(…) function.

sqrt((yade._minieigenHP.HP2.Real)x) -> yade._minieigenHP.HP2.Real :

return
`Real` square root of the argument. Depending on compilation options wraps `::boost::multiprecision::sqrt(…)` or std::sqrt(…) function.

`squaredNorm`(*(yade._minieigenHP.HP2.Complex)x*) → yade._minieigenHP.HP2.Real :

Returns
`Real` the norm (squared magnitude) of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::norm(…)` or std::norm(…) function.

`tan`(*(yade._minieigenHP.HP2.Complex)x*) → yade._minieigenHP.HP2.Complex :

Returns
`Complex` the tangent of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::tan(…)` or std::tan(…) function.

tan((yade._minieigenHP.HP2.Real)x) -> yade._minieigenHP.HP2.Real :

return
`Real` the tangent of the `Real` argument in radians. Depending on compilation options wraps `::boost::multiprecision::tan(…)` or std::tan(…) function.

`tanh`(*(yade._minieigenHP.HP2.Complex)x*) → yade._minieigenHP.HP2.Complex :

Returns
`Complex` the hyperbolic tangent of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::tanh(…)` or std::tanh(…) function.

tanh((yade._minieigenHP.HP2.Real)x) -> yade._minieigenHP.HP2.Real :

return
`Real` the hyperbolic tangent of the `Real` argument in radians. Depending on compilation options wraps `::boost::multiprecision::tanh(…)` or std::tanh(…) function.

`testArray`() → None :
This function tests call to `std::vector::data(…)` function in order to extract the array.

`testCgalNumTraits = True`

`testConstants`() → None :
This function tests lib/high-precision/Constants.hpp, the yade::math::ConstantsHP<N>, former yade::Mathr constants.

`tgamma`(*(yade._minieigenHP.HP2.Real)x*) → yade._minieigenHP.HP2.Real :

Returns
`Real` Computes the gamma function of arg. Depending on compilation options wraps `::boost::multiprecision::tgamma(…)` or std::tgamma(…) function.

`toDouble`(*(yade._minieigenHP.HP2.Real)x*) → float :

Returns
`float` converts `Real` type to `double` and returns a native python `float`.

`toHP1`(*(yade._minieigenHP.HP2.Real)x*) → float :

Returns
`RealHP<1>` converted from argument `RealHP<2>` as a result of `static_cast<RealHP<1>>(arg)`.

`toHP2`(*(yade._minieigenHP.HP2.Real)x*) → yade._minieigenHP.HP2.Real :

Returns
`RealHP<2>` converted from argument `RealHP<2>` as a result of `static_cast<RealHP<2>>(arg)`.

`toInt`(*(yade._minieigenHP.HP2.Real)x*) → int :

Returns
`int` converts `Real` type to `int` and returns a native python `int`.

`toLong`(*(yade._minieigenHP.HP2.Real)x*) → int :

Returns
`int` converts `Real` type to `long int` and returns a native python `int`.

`toLongDouble`(*(yade._minieigenHP.HP2.Real)x*) → float :

Returns
`float` converts `Real` type to `long double` and returns a native python `float`.

`trunc`(*(yade._minieigenHP.HP2.Real)x*) → yade._minieigenHP.HP2.Real :

Returns
`Real` the nearest integer not greater in magnitude than arg. Depending on compilation options wraps `::boost::multiprecision::trunc(…)` or std::trunc(…) function.

`yade._math.Log2`(*[(int)Precision=53]*) → float

Returns
`Real` natural logarithm of 2, exposed to python for testing of eigen numerical traits.

`yade._math.Pi`(*[(int)Precision=53]*) → float

Returns
`Real` The constant, exposed to python for testing of eigen numerical traits.

`class yade._math.RealHPConfig`

`RealHPConfig` class provides information about RealHP<N> type.

Variables

- `extraStringDigits10` – this static variable allows to control how many extra digits to use when converting to decimal strings. Assign a different value to it to affect the string conversion done in C++ python conversions as well as in all other conversions. Be careful, because values smaller than 3 can fail the round trip conversion test.
- `isFloat128Broken` – provides runtime information if Yade was compiled with g++ version < 9.2.1 and thus `boost::multiprecision::float128` cannot work.
- `isEnabledRealHP` – provides runtime information `RealHP<N>` is available for N higher than 1.
- `workaroundSlowBoostBinFloat` – `boost::multiprecision::cpp_bin_float` has some problem that importing it in python is very slow when these functions are exported: erf, erfc, lgamma, tgamma. In such case the python `import yade.math` can take more than minute. The workaround is to make them unavailable in python for higher N values. See invocation of IfConstexprForSlowFunctions in _math.cpp. This variable contains the highest N in which these functions are available. It equals to highest N when `boost::multiprecision::cpp_bin_float` is not used.

`extraStringDigits10 = 4`

`getDigits10`(*(int)N*) → int :

This is a yade.math.RealHPConfig diagnostic function.

Parameters
`N` – `int` - the value of `N` in `RealHP<N>`.

Returns
the `int` representing `std::numeric_limits<RealHP<N>>::digits10`

`getDigits2(`*(int)N*`)` → int :

This is a yade.math.RealHPConfig diagnostic function.

Parameters
`N` – `int` - the value of `N` in `RealHP<N>`.

Returns
the `int` representing `std::numeric_limits<RealHP<N>>::digits`, which corresponds to the number of significand bits used by this type.

`getSupportedByEigenCgal()` → tuple :

Returns
the `tuple` containing `N` from `RealHP<N>` precisions supported by Eigen and CGAL

`getSupportedByMinieigen()` → tuple :

Returns
the `tuple` containing `N` from `RealHP<N>` precisions supported by minieigenHP

`isEnabledRealHP = True`

`isFloat128Broken = False`

`isFloat128Present = True`

`workaroundSlowBoostBinFloat = 2`

`class` yade._math.`Var`

The `Var` class is used to test to/from python converters for arbitrary precision `Real`

`property cpl`

one `Complex` variable to test reading from and writing to it.

`property val`

one `Real` variable for testing.

yade._math.`abs(`*(complex)x*`)` → float

return
the `Real` absolute value of the `Complex` argument. Depending on compilation options wraps `::boost::multiprecision::abs(…)` or std::abs(…) function.

abs((float)x) → float :

return
the `Real` absolute value of the `Real` argument. Depending on compilation options wraps `::boost::multiprecision::abs(…)` or std::abs(…) function.

yade._math.`acos(`*(complex)x*`)` → complex

return
`Complex` the arc-cosine of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::acos(…)` or std::acos(…) function.

acos((float)x) → float :

return
`Real` the arcus cosine of the argument. Depending on compilation options wraps `::boost::multiprecision::acos(…)` or std::acos(…) function.

yade._math.acosh(*(complex)x*) → complex

return
`Complex` the arc-hyperbolic cosine of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::acosh(…)` or std::acosh(…) function.

acosh((float)x) → float :

return
`Real` the hyperbolic arcus cosine of the argument. Depending on compilation options wraps `::boost::multiprecision::acosh(…)` or std::acosh(…) function.

yade._math.arg(*(complex)x*) → float

Returns
`Real` the arg (Phase angle of complex in radians) of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::arg(…)` or std::arg(…) function.

yade._math.asin(*(complex)x*) → complex

return
`Complex` the arc-sine of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::asin(…)` or std::asin(…) function.

asin((float)x) → float :

return
`Real` the arcus sine of the argument. Depending on compilation options wraps `::boost::multiprecision::asin(…)` or std::asin(…) function.

yade._math.asinh(*(complex)x*) → complex

return
`Complex` the arc-hyperbolic sine of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::asinh(…)` or std::asinh(…) function.

asinh((float)x) → float :

return
`Real` the hyperbolic arcus sine of the argument. Depending on compilation options wraps `::boost::multiprecision::asinh(…)` or std::asinh(…) function.

yade._math.atan(*(complex)x*) → complex

return
`Complex` the arc-tangent of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::atan(…)` or std::atan(…) function.

atan((float)x) → float :

return
`Real` the arcus tangent of the argument. Depending on compilation options wraps `::boost::multiprecision::atan(…)` or std::atan(…) function.

yade._math.atan2(*(float)x*, *(float)y*) → float

Returns
Real the arc tangent of y/x using the signs of the arguments x and y to determine the correct quadrant. Depending on compilation options wraps ::boost::multiprecision::atan2(…) or std::atan2(…) function.

yade._math.atanh(*(complex)x*) → complex

return
Complex the arc-hyperbolic tangent of the Complex argument in radians. Depending on compilation options wraps ::boost::multiprecision::atanh(…) or std::atanh(…) function.

atanh((float)x) → float :

return
Real the hyperbolic arcus tangent of the argument. Depending on compilation options wraps ::boost::multiprecision::atanh(…) or std::atanh(…) function.

yade._math.cbrt(*(float)x*) → float

Returns
Real cubic root of the argument. Depending on compilation options wraps ::boost::multiprecision::cbrt(…) or std::cbrt(…) function.

yade._math.ceil(*(float)x*) → float

Returns
Real Computes the smallest integer value not less than arg. Depending on compilation options wraps ::boost::multiprecision::ceil(…) or std::ceil(…) function.

yade._math.conj(*(complex)x*) → complex

Returns
the complex conjugation a Complex argument. Depending on compilation options wraps ::boost::multiprecision::conj(…) or std::conj(…) function.

yade._math.cos(*(complex)x*) → complex

return
Complex the cosine of the Complex argument in radians. Depending on compilation options wraps ::boost::multiprecision::cos(…) or std::cos(…) function.

cos((float)x) → float :

return
Real the cosine of the Real argument in radians. Depending on compilation options wraps ::boost::multiprecision::cos(…) or std::cos(…) function.

yade._math.cosh(*(complex)x*) → complex

return
Complex the hyperbolic cosine of the Complex argument in radians. Depending on compilation options wraps ::boost::multiprecision::cosh(…) or std::cosh(…) function.

cosh((float)x) → float :

return
Real the hyperbolic cosine of the Real argument in radians. Depending on compilation options wraps ::boost::multiprecision::cosh(…) or std::cosh(…) function.

yade._math.cylBesselJ(*(int)k*, *(float)x*) → float

Returns
`Real` the Bessel Functions of the First Kind of the order `k` and the `Real` argument. See: <https://www.boost.org/doc/libs/1_77_0/libs/math/doc/html/math_toolkit/bessel/bessel_first.html>‘___

yade._math.dummy_precision() → float

Returns
similar to the function `epsilon`, but assumes that last 10% of bits contain the numerical error only. This is sometimes used by Eigen when calling `isEqualFuzzy` to determine if values differ a lot or if they are vaguely close to each other.

yade._math.epsilon(*[(int)Precision=53]*) → float

return
`Real` returns the difference between `1.0` and the next representable value of the `Real` type. Wraps std::numeric_limits<Real>::epsilon() function.

epsilon((float)x) → float :

return
`Real` returns the difference between `1.0` and the next representable value of the `Real` type. Wraps std::numeric_limits<Real>::epsilon() function.

yade._math.erf(*(float)x*) → float

Returns
`Real` Computes the error function of argument. Depending on compilation options wraps `::boost::multiprecision::erf(…)` or std::erf(…) function.

yade._math.erfc(*(float)x*) → float

Returns
`Real` Computes the complementary error function of argument, that is `1.0-erf(arg)`. Depending on compilation options wraps `::boost::multiprecision::erfc(…)` or std::erfc(…) function.

yade._math.exp(*(complex)x*) → complex

return
the base *e* exponential of a `Complex` argument. Depending on compilation options wraps `::boost::multiprecision::exp(…)` or std::exp(…) function.

exp((float)x) → float :

return
the base *e* exponential of a `Real` argument. Depending on compilation options wraps `::boost::multiprecision::exp(…)` or std::exp(…) function.

yade._math.exp2(*(float)x*) → float

Returns
the base *2* exponential of a `Real` argument. Depending on compilation options wraps `::boost::multiprecision::exp2(…)` or std::exp2(…) function.

yade._math.expm1(*(float)x*) → float

Returns
the base *e* exponential of a `Real` argument minus `1.0`. Depending on compilation options wraps `::boost::multiprecision::expm1(…)` or std::expm1(…) function.

yade._math.fabs(*(float)x*) → float

Returns
the Real absolute value of the argument. Depending on compilation options wraps ::boost::multiprecision::abs(…) or std::abs(…) function.

yade._math.factorial(*(int)x*) → float

Returns
Real the factorial of the Real argument. See: <https://www.boost.org/doc/libs/1_77_0/libs/math/doc/html/math_toolkit/factorials/sf_factorial.html>‘___

yade._math.floor(*(float)x*) → float

Returns
Real Computes the largest integer value not greater than arg. Depending on compilation options wraps ::boost::multiprecision::floor(…) or std::floor(…) function.

yade._math.fma(*(float)x*, *(float)y*, *(float)z*) → float

Returns
Real - computes (x*y) + z as if to infinite precision and rounded only once to fit the result type. Depending on compilation options wraps ::boost::multiprecision::fma(…) or std::fma(…) function.

yade._math.fmod(*(float)x*, *(float)y*) → float

Returns
Real the floating-point remainder of the division operation x/y of the arguments x and y. Depending on compilation options wraps ::boost::multiprecision::fmod(…) or std::fmod(…) function.

yade._math.frexp(*(float)x*) → tuple

Returns
tuple of (Real,int), decomposes given floating point Real argument into a normalized fraction and an integral power of two. Depending on compilation options wraps ::boost::multiprecision::frexp(…) or std::frexp(…) function.

yade._math.fromBits(*(str)bits*[, *(int)exp=0*[, *(int)sign=1*]])) → float

Parameters

- **bits** – str - a string containing ‘0’, ‘1’ characters.
- **exp** – int - the binary exponent which shifts the bits.
- **sign** – int - the sign, should be -1 or +1, but it is not checked. It multiplies the result when construction from bits is finished.

Returns
RealHP<N> constructed from string containing ‘0’, ‘1’ bits. This is for debugging purposes, rather slow.

yade._math.getDecomposedReal(*(float)x*) → dict

Returns
dict - the dictionary with the debug information how the DecomposedReal class sees this type. This is for debugging purposes, rather slow. Includes result from fpclassify function call, a binary representation and other useful info. See also *fromBits*.

yade._math.getDemangledName() → str

Returns
string - the demangled C++ typnename of RealHP<N>.

yade._math.getDemangledNameComplex() → str

Returns
`string` - the demangled C++ typnename of `ComplexHP<N>`.

yade._math.getEigenFlags() → dict

Returns
A python dictionary listing flags for all types, see: https://eigen.tuxfamily.org/dox/group___flags.html

yade._math.getEigenStorageOrders() → dict

Returns
A python dictionary listing options for all types, see: https://eigen.tuxfamily.org/dox/group___TopicStorageOrders.html

yade._math.getFloatDistanceULP(*(float)arg1*, *(float)arg2*) → float

Returns
an integer value stored in `RealHP<N>`, the ULP distance calculated by boost::math::float_distance, also see Floating-point Comparison and Prof. Kahan paper about this topic.

The returned value is the **directed distance** between two arguments, this means that it can be negative.

yade._math.getRawBits(*(float)x*) → str

Returns
`string` - the raw bits in memory representing this type. Be careful: it only checks the system endianness and either prints bytes in reverse order or not. Does not make any attempts to further interpret the bits of: sign, exponent or significand (on a typical x86 processor they are printed in that order), and different processors might store them differently. It is not useful for types which internally use a pointer because for them this function prints not the floating point number but a pointer. This is for debugging purposes.

yade._math.getRealHPErrors(*(list)testLevelsHP*[, *(int)testCount=10*[, *(float)minX=-10.0*[, *(float)maxX=10.0*[, *(bool)useRandomArgs=False*[, *(int)printEveryNth=1000*[, *(bool)collectArgs=False*[, *(bool)extraChecks=False*]]]]]]]) → dict

Tests mathematical functions against the highest precision in argument `testLevelsHP` and returns the largest ULP distance found with *getFloatDistanceULP*. A `testCount` randomized tries with function arguments in range `minX ... maxX` are performed on the `RealHP<N>` types where `N` is from the list provided in `testLevelsHP` argument.

Parameters

- **testLevelsHP** – a `list` of `int` values consisting of high precision levels `N` (in `RealHP<N>`) for which the tests should be done. Must consist at least of two elements so that there is a higher precision type available against which to perform the tests.
- **testCount** – `int` - specifies how many randomized tests of each function to perform.
- **minX** – `Real` - start of the range in which the random arguments are generated.
- **maxX** – `Real` - end of that range.
- **useRandomArgs** – If `False` (default) then `minX ... maxX` is divided into `testCount` equidistant points. If `True` then each call is a random number. This

applies only to the first argument of a function, if a function takes more than one argument, then remaining arguments are random - 2D scans are not performed.

- **printEveryNth** – will *print using* LOG_INFO the progress information every Nth step in the testCount loop. To see it e.g. start using yade -f6, also see *logger documentation*.
- **collectArgs** – if True then in returned results will be a longer list of arguments that produce incorrect results.
- **extraChecks** – will perform extra checks while executing this funcion. Useful only for debugging of *getRealHPErrors*.

Returns

A python dictionary with the largest ULP distance to the correct function value. For each function name there is a dictionary consisting of: how many binary digits (bits) are in the tested RealHP<N> type, the worst arguments for this function, and the ULP distance to the reference value.

The returned ULP error is an absolute value, as opposed to *getFloatDistanceULP* which is signed.

yade._math.**highest**([*(int)Precision=53*]) → float

Returns

Real returns the largest finite value of the Real type. Wraps std::numeric_limits<Real>::max() function.

yade._math.**hypot**(*(float)x*, *(float)y*) → float

Returns

Real the square root of the sum of the squares of x and y, without undue overflow or underflow at intermediate stages of the computation. Depending on compilation options wraps ::boost::multiprecision::hypot(…) or std::hypot(…) function.

yade._math.**ilogb**(*(float)x*) → float

Returns

Real extracts the value of the unbiased exponent from the floating-point argument arg, and returns it as a signed integer value. Depending on compilation options wraps ::boost::multiprecision::ilogb(…) or std::ilogb(…) function.

yade._math.**imag**(*(complex)x*) → float

Returns

the imag part of a Complex argument. Depending on compilation options wraps ::boost::multiprecision::imag(…) or std::imag(…) function.

yade._math.**isApprox**(*(float)a*, *(float)b*, *(float)eps*) → bool

Returns

bool, True if a is approximately equal b with provided eps, see also here

yade._math.**isApproxOrLessThan**(*(float)a*, *(float)b*, *(float)eps*) → bool

Returns

bool, True if a is approximately less than or equal b with provided eps, see also here

yade._math.**isEqualFuzzy**(*(float)arg1*, *(float)arg2*, *(float)arg3*) → bool

Returns

bool, True if the absolute difference between two numbers is smaller than std::numeric_limits<Real>::epsilon()

yade._math.**isMuchSmallerThan**(*(float)a*, *(float)b*, *(float)eps*) → bool

Returns

bool, True if a is less than b with provided eps, see also here

yade._math.isThisSystemLittleEndian() → bool

Returns
`True` if this system uses little endian architecture, `False` otherwise.

yade._math.isfinite(*(float)x*) → bool

Returns
`bool` indicating if the `Real` argument is Inf. Depending on compilation options wraps `::boost::multiprecision::isfinite(…)` or std::isfinite(…) function.

yade._math.isinf(*(float)x*) → bool

Returns
`bool` indicating if the `Real` argument is Inf. Depending on compilation options wraps `::boost::multiprecision::isinf(…)` or std::isinf(…) function.

yade._math.isnan(*(float)x*) → bool

Returns
`bool` indicating if the `Real` argument is NaN. Depending on compilation options wraps `::boost::multiprecision::isnan(…)` or std::isnan(…) function.

yade._math.laguerre(*(int)n*, *(int)m*, *(float)x*) → float

Returns
`Real` the Laguerre polynomial of the orders `n`, `m` and the `Real` argument. See: <https://www.boost.org/doc/libs/1_77_0/libs/math/doc/html/math_toolkit/sf_poly/laguerre.html>‘___

yade._math.ldexp(*(float)x*, *(int)y*) → float

Returns
Multiplies a floating point value `x` by the number 2 raised to the `exp` power. Depending on compilation options wraps `::boost::multiprecision::ldexp(…)` or std::ldexp(…) function.

yade._math.lgamma(*(float)x*) → float

Returns
`Real` Computes the natural logarithm of the absolute value of the gamma function of arg. Depending on compilation options wraps `::boost::multiprecision::lgamma(…)` or std::lgamma(…) function.

yade._math.log(*(complex)x*) → complex

return
the `Complex` natural (base *e*) logarithm of a complex value z with a branch cut along the negative real axis. Depending on compilation options wraps `::boost::multiprecision::log(…)` or std::log(…) function.

log((float)x) → float :

return
the `Real` natural (base *e*) logarithm of a real value. Depending on compilation options wraps `::boost::multiprecision::log(…)` or std::log(…) function.

yade._math.log10(*(complex)x*) → complex

return
the `Complex` (base *10*) logarithm of a complex value z with a branch cut along the negative real axis. Depending on compilation options wraps `::boost::multiprecision::log10(…)` or std::log10(…) function.

log10((float)x) → float :

return
the `Real` decimal (base `10`) logarithm of a real value. Depending on compilation options wraps `::boost::multiprecision::log10(…)` or std::log10(…) function.

`yade._math.log1p(`*(float)x*`)` → float

Returns
the `Real` natural (base *e*) logarithm of `1+argument`. Depending on compilation options wraps `::boost::multiprecision::log1p(…)` or std::log1p(…) function.

`yade._math.log2(`*(float)x*`)` → float

Returns
the `Real` binary (base `2`) logarithm of a real value. Depending on compilation options wraps `::boost::multiprecision::log2(…)` or std::log2(…) function.

`yade._math.logb(`*(float)x*`)` → float

Returns
Extracts the value of the unbiased radix-independent exponent from the floating-point argument arg, and returns it as a floating-point value. Depending on compilation options wraps `::boost::multiprecision::logb(…)` or std::logb(…) function.

`yade._math.lowest(`[*(int)Precision=53*]`)` → float

Returns
`Real` returns the lowest (negative) finite value of the `Real` type. Wraps std::numeric_limits<Real>::lowest() function.

`yade._math.max(`*(float)x*, *(float)y*`)` → float

Returns
`Real` larger of the two arguments. Depending on compilation options wraps `::boost::multiprecision::max(…)` or std::max(…) function.

`yade._math.min(`*(float)x*, *(float)y*`)` → float

Returns
`Real` smaller of the two arguments. Depending on compilation options wraps `::boost::multiprecision::min(…)` or std::min(…) function.

`yade._math.modf(`*(float)x*`)` → tuple

Returns
tuple of `(Real,Real)`, decomposes given floating point `Real` into integral and fractional parts, each having the same type and sign as x. Depending on compilation options wraps `::boost::multiprecision::modf(…)` or std::modf(…) function.

`yade._math.polar(`*(float)x*, *(float)y*`)` → complex

Returns
`Complex` the polar (Complex from polar components) of the `Real` rho (length), `Real` theta (angle) arguments in radians. Depending on compilation options wraps `::boost::multiprecision::polar(…)` or std::polar(…) function.

`yade._math.pow(`*(complex)x*, *(complex)pow*`)` → complex

return
the `Complex` complex arg1 raised to the `Complex` power arg2. Depending on compilation options wraps `::boost::multiprecision::pow(…)` or std::pow(…) function.

pow((float)x, (float)y) → float :

return
`Real` the value of `base` raised to the power `exp`. Depending on compilation options wraps `::boost::multiprecision::pow(…)` or std::pow(…) function.

`yade._math.proj(`*(complex)x*`)` → complex

Returns
`Complex` the proj (projection of the complex number onto the Riemann sphere) of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::proj(…)` or std::proj(…) function.

`yade._math.random()` → float

return
`Real` a symmetric random number in interval `(-1,1)`. Used by Eigen.

random((float)a, (float)b) → float :

return
`Real` a random number in interval `(a,b)`. Used by Eigen.

`yade._math.real(`*(complex)x*`)` → float

Returns
the real part of a `Complex` argument. Depending on compilation options wraps `::boost::multiprecision::real(…)` or std::real(…) function.

`yade._math.remainder(`*(float)x*, *(float)y*`)` → float

Returns
`Real` the IEEE remainder of the floating point division operation `x/y`. Depending on compilation options wraps `::boost::multiprecision::remainder(…)` or std::remainder(…) function.

`yade._math.remquo(`*(float)x*, *(float)y*`)` → tuple

Returns
tuple of `(Real,long)`, the floating-point remainder of the division operation `x/y` as the std::remainder() function does. Additionally, the sign and at least the three of the last bits of `x/y` are returned, sufficient to determine the octant of the result within a period. Depending on compilation options wraps `::boost::multiprecision::remquo(…)` or std::remquo(…) function.

`yade._math.rint(`*(float)x*`)` → float

Returns
Rounds the floating-point argument arg to an integer value (in floating-point format), using the current rounding mode. Depending on compilation options wraps `::boost::multiprecision::rint(…)` or std::rint(…) function.

`yade._math.round(`*(float)x*`)` → float

Returns
`Real` the nearest integer value to arg (in floating-point format), rounding halfway cases away from zero, regardless of the current rounding mode.. Depending on compilation options wraps `::boost::multiprecision::round(…)` or std::round(…) function.

`yade._math.roundTrip(`*(float)x*`)` → float

Returns
`Real` returns the argument `x`. Can be used to convert type to native RealHP<N> accuracy.

yade._math.**sgn**(*(float)x*) → int

Returns
`int` the sign of the argument: `-1`, `0` or `1`.

yade._math.**sign**(*(float)x*) → int

Returns
`int` the sign of the argument: `-1`, `0` or `1`.

yade._math.**sin**(*(complex)x*) → complex

return
`Complex` the sine of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::sin(…)` or std::sin(…) function.

sin((float)x) → float :

return
`Real` the sine of the `Real` argument in radians. Depending on compilation options wraps `::boost::multiprecision::sin(…)` or std::sin(…) function.

yade._math.**sinh**(*(complex)x*) → complex

return
`Complex` the hyperbolic sine of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::sinh(…)` or std::sinh(…) function.

sinh((float)x) → float :

return
`Real` the hyperbolic sine of the `Real` argument in radians. Depending on compilation options wraps `::boost::multiprecision::sinh(…)` or std::sinh(…) function.

yade._math.**smallest_positive**() → float

Returns
`Real` the smallest number greater than zero. Wraps std::numeric_limits<Real>::min()

yade._math.**sphericalHarmonic**(*(int)l*, *(int)m*, *(float)theta*, *(float)phi*) → complex

Returns
`Real` the spherical harmonic polynomial of the orders `l` (unsigned int), `m` (signed int) and the `Real` arguments `theta` and `phi`. See: <https://www.boost.org/doc/libs/1_77_0/libs/math/doc/html/math_toolkit/sf_poly/sph_harm.html>‘___

yade._math.**sqrt**(*(complex)x*) → complex

return
the `Complex` square root of `Complex` argument. Depending on compilation options wraps `::boost::multiprecision::sqrt(…)` or std::sqrt(…) function.

sqrt((float)x) → float :

return
`Real` square root of the argument. Depending on compilation options wraps `::boost::multiprecision::sqrt(…)` or std::sqrt(…) function.

yade._math.squaredNorm(*(complex)x*) → float

Returns
`Real` the norm (squared magnitude) of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::norm(…)` or std::norm(…) function.

yade._math.tan(*(complex)x*) → complex

return
`Complex` the tangent of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::tan(…)` or std::tan(…) function.

tan((float)x) → float :

return
`Real` the tangent of the `Real` argument in radians. Depending on compilation options wraps `::boost::multiprecision::tan(…)` or std::tan(…) function.

yade._math.tanh(*(complex)x*) → complex

return
`Complex` the hyperbolic tangent of the `Complex` argument in radians. Depending on compilation options wraps `::boost::multiprecision::tanh(…)` or std::tanh(…) function.

tanh((float)x) → float :

return
`Real` the hyperbolic tangent of the `Real` argument in radians. Depending on compilation options wraps `::boost::multiprecision::tanh(…)` or std::tanh(…) function.

yade._math.testArray() → None
This function tests call to `std::vector::data(…)` function in order to extract the array.

yade._math.testConstants() → None
This function tests lib/high-precision/Constants.hpp, the yade::math::ConstantsHP<N>, former yade::Mathr constants.

yade._math.testLoopRealHP() → None
This function tests lib/high-precision/Constants.hpp, but the C++ side: all precisions, even those inaccessible from python

yade._math.tgamma(*(float)x*) → float

Returns
`Real` Computes the gamma function of arg. Depending on compilation options wraps `::boost::multiprecision::tgamma(…)` or std::tgamma(…) function.

yade._math.toDouble(*(float)x*) → float

Returns
`float` converts `Real` type to `double` and returns a native python `float`.

yade._math.toHP1(*(float)x*) → float

Returns
`RealHP<1>` converted from argument `RealHP<1>` as a result of `static_cast<RealHP<1>>(arg)`.

yade._math.toHP2(*(float)x*) → *yade._minieigenHP.HP2.Real*

Returns
RealHP<2> converted from argument RealHP<1> as a result of static_cast<RealHP<2>>(arg).

yade._math.toInt(*(float)x*) → int

Returns
int converts Real type to int and returns a native python int.

yade._math.toLong(*(float)x*) → int

Returns
int converts Real type to long int and returns a native python int.

yade._math.toLongDouble(*(float)x*) → float

Returns
float converts Real type to long double and returns a native python float.

yade._math.trunc(*(float)x*) → float

Returns
Real the nearest integer not greater in magnitude than arg. Depending on compilation options wraps ::boost::multiprecision::trunc(…) or std::trunc(…) function.

2.4.10 yade.minieigenHP module

When yade uses high-precision number as Real type the usual (old):

```
from minieigen import *
```

has to be replaced with:

```
from yade.minieigenHP import *
```

This command ensures backward compatibility between both. It is then guaranteed that python uses the same number of decimal places as yade is using everywhere else.

Please note that used precision can be very arbitrary, because cpp_bin_float or mpfr take it as a *compile-time argument*. Hence such yade.minieigenHP cannot be separately precompiled as a package. Though it could be precompiled for some special types such as boost::multiprecision::float128.

The RealHP<n> *higher precision* vectors and matrices can be accessed in python by using the .HPn module scope. For example:

```
import yade.minieigenHP as mne
mne.HP2.Vector3(1,2,3) # produces Vector3 using RealHP<2> precision
mne.Vector3(1,2,3)     # without using HPn module scope it defaults to RealHP<1>
```

miniEigen is wrapper for a small part of the Eigen library. Refer to its documentation for details. All classes in this module support pickling.

class yade._minieigenHP.AlignedBox2

Axis-aligned box object in 2d, defined by its minimum and maximum corners

__init__(*(object)arg1*) → None

___init___((object)arg1, (AlignedBox2)other) -> None

___init___((object)arg1, (Vector2)min, (Vector2)max) -> None

center(*(AlignedBox2)arg1*) → *Vector2*

clamp(*(AlignedBox2)arg1*, *(AlignedBox2)arg2*) → None

contains(*(AlignedBox2)arg1*, *(Vector2)arg2*) → bool
contains((AlignedBox2)arg1, (AlignedBox2)arg2) -> bool

empty(*(AlignedBox2)arg1*) → bool

extend(*(AlignedBox2)arg1*, *(Vector2)arg2*) → None
extend((AlignedBox2)arg1, (AlignedBox2)arg2) -> None

intersection(*(AlignedBox2)arg1*, *(AlignedBox2)arg2*) → *AlignedBox2*

property max
None((yade._minieigenHP.AlignedBox2)arg1) -> yade._minieigenHP.Vector2

merged(*(AlignedBox2)arg1*, *(AlignedBox2)arg2*) → *AlignedBox2*

property min
None((yade._minieigenHP.AlignedBox2)arg1) -> yade._minieigenHP.Vector2

sizes(*(AlignedBox2)arg1*) → *Vector2*

volume(*(AlignedBox2)arg1*) → float

class yade._minieigenHP.**AlignedBox3**
Axis-aligned box object, defined by its minimum and maximum corners

__init__(*(object)arg1*) → None
___init___((object)arg1, (AlignedBox3)other) -> None
___init___((object)arg1, (Vector3)min, (Vector3)max) -> None

center(*(AlignedBox3)arg1*) → *Vector3*

clamp(*(AlignedBox3)arg1*, *(AlignedBox3)arg2*) → None

contains(*(AlignedBox3)arg1*, *(Vector3)arg2*) → bool
contains((AlignedBox3)arg1, (AlignedBox3)arg2) -> bool

empty(*(AlignedBox3)arg1*) → bool

extend(*(AlignedBox3)arg1*, *(Vector3)arg2*) → None
extend((AlignedBox3)arg1, (AlignedBox3)arg2) -> None

intersection(*(AlignedBox3)arg1*, *(AlignedBox3)arg2*) → *AlignedBox3*

property max
None((yade._minieigenHP.AlignedBox3)arg1) -> yade._minieigenHP.Vector3

merged(*(AlignedBox3)arg1*, *(AlignedBox3)arg2*) → *AlignedBox3*

property min
None((yade._minieigenHP.AlignedBox3)arg1) -> yade._minieigenHP.Vector3

sizes(*(AlignedBox3)arg1*) → *Vector3*

volume(*(AlignedBox3)arg1*) → float

class yade._minieigenHP.**HP1**

class AlignedBox2
Axis-aligned box object in 2d, defined by its minimum and maximum corners

__init__(*(object)arg1*) → None
___init___((object)arg1, (AlignedBox2)other) -> None
___init___((object)arg1, (Vector2)min, (Vector2)max) -> None

center(*(AlignedBox2)arg1*) → *Vector2*

clamp(*(AlignedBox2)arg1*, *(AlignedBox2)arg2*) → None

contains(*(AlignedBox2)arg1*, *(Vector2)arg2*) → bool
contains((AlignedBox2)arg1, (AlignedBox2)arg2) -> bool

empty(*(AlignedBox2)arg1*) → bool

extend(*(AlignedBox2)arg1*, *(Vector2)arg2*) → None
extend((AlignedBox2)arg1, (AlignedBox2)arg2) -> None

intersection(*(AlignedBox2)arg1*, *(AlignedBox2)arg2*) → *AlignedBox2*

property max
None((yade._minieigenHP.AlignedBox2)arg1) -> yade._minieigenHP.Vector2

merged(*(AlignedBox2)arg1*, *(AlignedBox2)arg2*) → *AlignedBox2*

property min
None((yade._minieigenHP.AlignedBox2)arg1) -> yade._minieigenHP.Vector2

sizes(*(AlignedBox2)arg1*) → *Vector2*

volume(*(AlignedBox2)arg1*) → float

class AlignedBox3
Axis-aligned box object, defined by its minimum and maximum corners

__init__(*(object)arg1*) → None
___init___((object)arg1, (AlignedBox3)other) -> None
___init___((object)arg1, (Vector3)min, (Vector3)max) -> None

center(*(AlignedBox3)arg1*) → *Vector3*

clamp(*(AlignedBox3)arg1*, *(AlignedBox3)arg2*) → None

contains(*(AlignedBox3)arg1*, *(Vector3)arg2*) → bool
contains((AlignedBox3)arg1, (AlignedBox3)arg2) -> bool

empty(*(AlignedBox3)arg1*) → bool

extend(*(AlignedBox3)arg1*, *(Vector3)arg2*) → None
extend((AlignedBox3)arg1, (AlignedBox3)arg2) -> None

intersection(*(AlignedBox3)arg1*, *(AlignedBox3)arg2*) → *AlignedBox3*

property max
None((yade._minieigenHP.AlignedBox3)arg1) -> yade._minieigenHP.Vector3

merged(*(AlignedBox3)arg1*, *(AlignedBox3)arg2*) → *AlignedBox3*

property min
None((yade._minieigenHP.AlignedBox3)arg1) -> yade._minieigenHP.Vector3

sizes(*(AlignedBox3)arg1*) → *Vector3*

volume(*(AlignedBox3)arg1*) → float

`Complex`

alias of `complex`

`class Matrix3`

3x3 float matrix.

Supported operations (`m` is a Matrix3, `f` if a float/int, `v` is a Vector3): `-m`, `m+m`, `m+=m`, `m-m`, `m-=m`, `m*f`, `f*m`, `m*=f`, `m/f`, `m/=f`, `m*m`, `m*=m`, `m*v`, `v*m`, `m==m`, `m!=m`.

Static attributes: `Zero`, `Ones`, `Identity`.

```
Identity = Matrix3(1,0,0, 0,1,0, 0,0,1)
```

```
Ones = Matrix3(1,1,1, 1,1,1, 1,1,1)
```

`static Random()` → Matrix3 :

Return an object where all elements are randomly set to values between 0 and 1.

```
Zero = Matrix3(0,0,0, 0,0,0, 0,0,0)
```

`__init__(`*(object)arg1*`)` → None

___init___((object)arg1, (Quaternion)q) -> None

___init___((object)arg1, (Matrix3)other) -> None

___init___((object)arg1, (Vector3)diag) -> object

___init___((object)arg1, (float)m00, (float)m01, (float)m02, (float)m10, (float)m11, (float)m12, (float)m20, (float)m21, (float)m22) -> object

___init___((object)arg1, (str)m00, (str)m01, (str)m02, (str)m10, (str)m11, (str)m12, (str)m20, (str)m21, (str)m22) -> object

___init___((object)arg1, (Vector3)r0, (Vector3)r1, (Vector3)r2 [, (bool)cols=False]) -> object

`col(`*(Matrix3)arg1*, *(int)col*`)` → Vector3 :

Return column as vector.

`cols(`*(Matrix3)arg1*`)` → int :

Number of columns.

`computeUnitaryPositive(`*(Matrix3)arg1*`)` → tuple :

Compute polar decomposition (unitary matrix U and positive semi-definite symmetric matrix P such that self=U*P).

`determinant(`*(Matrix3)arg1*`)` → float :

Return matrix determinant.

`diagonal(`*(Matrix3)arg1*`)` → Vector3 :

Return diagonal as vector.

`inverse(`*(Matrix3)arg1*`)` → Matrix3 :

Return inverted matrix.

`isApprox(`*(Matrix3)arg1*, *(Matrix3)other*[, *(float)prec=1e-12*]`)` → bool :

Approximate comparison with precision *prec*.

`jacobiSVD(`*(Matrix3)arg1*`)` → tuple :

Compute SVD decomposition of square matrix, retuns (U,S,V) such that self=U*S*V.transpose()

`maxAbsCoeff(`*(Matrix3)arg1*`)` → float :

Maximum absolute value over all elements.

maxCoeff(*(Matrix3)arg1*) → float :

Maximum value over all elements.

mean(*(Matrix3)arg1*) → float :

Mean value over all elements.

minCoeff(*(Matrix3)arg1*) → float :

Minimum value over all elements.

norm(*(Matrix3)arg1*) → float :

Euclidean norm.

normalize(*(Matrix3)arg1*) → None :

Normalize this object in-place.

normalized(*(Matrix3)arg1*) → Matrix3 :

Return normalized copy of this object

polarDecomposition(*(Matrix3)arg1*) → tuple :

Alias for *computeUnitaryPositive*.

prod(*(Matrix3)arg1*) → float :

Product of all elements.

pruned(*(Matrix3)arg1*[, *(float)absTol=1e-06*]) → Matrix3 :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

row(*(Matrix3)arg1*, *(int)row*) → Vector3 :

Return row as vector.

rows(*(Matrix3)arg1*) → int :

Number of rows.

selfAdjointEigenDecomposition(*(Matrix3)arg1*) → tuple :

Compute eigen (spectral) decomposition of symmetric matrix, returns (eigVecs,eigVals). eigVecs is orthogonal Matrix3 with columns ar normalized eigenvectors, eigVals is Vector3 with corresponding eigenvalues. self=eigVecs*diag(eigVals)*eigVecs.transpose().

spectralDecomposition(*(Matrix3)arg1*) → tuple :

Alias for *selfAdjointEigenDecomposition*.

squaredNorm(*(Matrix3)arg1*) → float :

Square of the Euclidean norm.

sum(*(Matrix3)arg1*) → float :

Sum of all elements.

svd(*(Matrix3)arg1*) → tuple :

Alias for *jacobiSVD*.

trace(*(Matrix3)arg1*) → float :

Return sum of diagonal elements.

transpose(*(Matrix3)arg1*) → Matrix3 :

Return transposed matrix.

class Matrix3c

/TODO/

```
Identity = Matrix3c(1,0,0, 0,1,0, 0,0,1)
```

```
Ones = Matrix3c(1,1,1, 1,1,1, 1,1,1)
```

static Random() → Matrix3c :

Return an object where all elements are randomly set to values between 0 and 1.

Zero = Matrix3c(0,0,0, 0,0,0, 0,0,0)

__init__(*(object)arg1*) → None

___init___((object)arg1, (Matrix3c)other) -> None

___init___((object)arg1, (Vector3c)diag) -> object

___init___((object)arg1, (complex)m00, (complex)m01, (complex)m02, (complex)m10, (complex)m11, (complex)m12, (complex)m20, (complex)m21, (complex)m22) -> object

___init___((object)arg1, (str)m00, (str)m01, (str)m02, (str)m10, (str)m11, (str)m12, (str)m20, (str)m21, (str)m22) -> object

___init___((object)arg1, (Vector3c)r0, (Vector3c)r1, (Vector3c)r2 [, (bool)cols=False]) -> object

col(*(Matrix3c)arg1*, *(int)col*) → Vector3c :

Return column as vector.

cols(*(Matrix3c)arg1*) → int :

Number of columns.

determinant(*(Matrix3c)arg1*) → complex :

Return matrix determinant.

diagonal(*(Matrix3c)arg1*) → Vector3c :

Return diagonal as vector.

inverse(*(Matrix3c)arg1*) → Matrix3c :

Return inverted matrix.

isApprox(*(Matrix3c)arg1*, *(Matrix3c)other*[, *(float)prec=1e-12*]) → bool :

Approximate comparison with precision *prec*.

maxAbsCoeff(*(Matrix3c)arg1*) → float :

Maximum absolute value over all elements.

mean(*(Matrix3c)arg1*) → complex :

Mean value over all elements.

norm(*(Matrix3c)arg1*) → float :

Euclidean norm.

normalize(*(Matrix3c)arg1*) → None :

Normalize this object in-place.

normalized(*(Matrix3c)arg1*) → Matrix3c :

Return normalized copy of this object

prod(*(Matrix3c)arg1*) → complex :

Product of all elements.

pruned(*(Matrix3c)arg1*[, *(float)absTol=1e-06*]) → Matrix3c :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

row(*(Matrix3c)arg1*, *(int)row*) → Vector3c :

Return row as vector.

rows(*(Matrix3c)arg1*) → int :

Number of rows.

squaredNorm(*(Matrix3c)arg1*) → float :

Square of the Euclidean norm.

sum(*(Matrix3c)arg1*) → complex :

Sum of all elements.

trace(*(Matrix3c)arg1*) → complex :

Return sum of diagonal elements.

transpose(*(Matrix3c)arg1*) → Matrix3c :

Return transposed matrix.

class Matrix6

6x6 float matrix. Constructed from 4 3x3 sub-matrices, from 6xVector6 (rows).

Supported operations (`m` is a Matrix6, `f` if a float/int, `v` is a Vector6): `-m`, `m+m`, `m+=m`, `m-m`, `m-=m`, `m*f`, `f*m`, `m*=f`, `m/f`, `m/=f`, `m*m`, `m*=m`, `m*v`, `v*m`, `m==m`, `m!=m`.

Static attributes: `Zero`, `Ones`, `Identity`.

Identity = Matrix6((1,0,0,0,0,0), (0,1,0,0,0,0), (0,0,1,0,0,0), (0,0,0,1,0,0), (0,0,0,0,1,0), (0,0,0,0,0,1))

Ones = Matrix6((1,1,1,1,1,1), (1,1,1,1,1,1), (1,1,1,1,1,1), (1,1,1,1,1,1), (1,1,1,1,1,1), (1,1,1,1,1,1))

static Random() → Matrix6 :

Return an object where all elements are randomly set to values between 0 and 1.

Zero = Matrix6((0,0,0,0,0,0), (0,0,0,0,0,0), (0,0,0,0,0,0), (0,0,0,0,0,0), (0,0,0,0,0,0), (0,0,0,0,0,0))

__init__(*(object)arg1*) → None

___init___((object)arg1, (Matrix6)other) -> None

___init___((object)arg1, (Vector6)diag) -> object

___init___((object)arg1, (Matrix3)ul, (Matrix3)ur, (Matrix3)ll, (Matrix3)lr) -> object

___init___((object)arg1, (Vector6)l0, (Vector6)l1, (Vector6)l2, (Vector6)l3, (Vector6)l4, (Vector6)l5 [, (bool)cols=False]) -> object

col(*(Matrix6)arg1*, *(int)col*) → Vector6 :

Return column as vector.

cols(*(Matrix6)arg1*) → int :

Number of columns.

computeUnitaryPositive(*(Matrix6)arg1*) → tuple :

Compute polar decomposition (unitary matrix U and positive semi-definite symmetric matrix P such that self=U*P).

determinant(*(Matrix6)arg1*) → float :

Return matrix determinant.

diagonal(*(Matrix6)arg1*) → Vector6 :

Return diagonal as vector.

inverse(*(Matrix6)arg1*) → Matrix6 :

Return inverted matrix.

isApprox(*(Matrix6)arg1*, *(Matrix6)other*[, *(float)prec=1e-12*]) → bool :

Approximate comparison with precision *prec*.

jacobiSVD(*(Matrix6)arg1*) → tuple :

Compute SVD decomposition of square matrix, retuns (U,S,V) such that self=U*S*V.transpose()

ll(*(Matrix6)arg1*) → Matrix3 :

Return lower-left 3x3 block

lr(*(Matrix6)arg1*) → Matrix3 :

Return lower-right 3x3 block

maxAbsCoeff(*(Matrix6)arg1*) → float :

Maximum absolute value over all elements.

maxCoeff(*(Matrix6)arg1*) → float :

Maximum value over all elements.

mean(*(Matrix6)arg1*) → float :

Mean value over all elements.

minCoeff(*(Matrix6)arg1*) → float :

Minimum value over all elements.

norm(*(Matrix6)arg1*) → float :

Euclidean norm.

normalize(*(Matrix6)arg1*) → None :

Normalize this object in-place.

normalized(*(Matrix6)arg1*) → Matrix6 :

Return normalized copy of this object

polarDecomposition(*(Matrix6)arg1*) → tuple :

Alias for *computeUnitaryPositive*.

prod(*(Matrix6)arg1*) → float :

Product of all elements.

pruned(*(Matrix6)arg1*[, *(float)absTol=1e-06*]) → Matrix6 :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

row(*(Matrix6)arg1*, *(int)row*) → Vector6 :

Return row as vector.

rows(*(Matrix6)arg1*) → int :

Number of rows.

selfAdjointEigenDecomposition(*(Matrix6)arg1*) → tuple :

Compute eigen (spectral) decomposition of symmetric matrix, returns (eigVecs,eigVals). eigVecs is orthogonal Matrix3 with columns ar normalized eigenvectors, eigVals is Vector3 with corresponding eigenvalues. self=eigVecs*diag(eigVals)*eigVecs.transpose().

spectralDecomposition(*(Matrix6)arg1*) → tuple :

Alias for *selfAdjointEigenDecomposition*.

squaredNorm(*(Matrix6)arg1*) → float :

Square of the Euclidean norm.

sum(*(Matrix6)arg1*) → float :

Sum of all elements.

svd(*(Matrix6)arg1*) → tuple :

Alias for *jacobiSVD*.

`trace`(*(Matrix6)arg1*) → float :

Return sum of diagonal elements.

`transpose`(*(Matrix6)arg1*) → Matrix6 :

Return transposed matrix.

`ul`(*(Matrix6)arg1*) → Matrix3 :

Return upper-left 3x3 block

`ur`(*(Matrix6)arg1*) → Matrix3 :

Return upper-right 3x3 block

`class Matrix6c`

/TODO/

`Identity = Matrix6c( (1,0,0,0,0,0), (0,1,0,0,0,0), (0,0,1,0,0,0), (0,0,0,1,0,0), (0,0,0,0,1,0), (0,0,0,0,0,1) )`

`Ones = Matrix6c( (1,1,1,1,1,1), (1,1,1,1,1,1), (1,1,1,1,1,1), (1,1,1,1,1,1), (1,1,1,1,1,1), (1,1,1,1,1,1) )`

`static Random()` → Matrix6c :

Return an object where all elements are randomly set to values between 0 and 1.

`Zero = Matrix6c( (0,0,0,0,0,0), (0,0,0,0,0,0), (0,0,0,0,0,0), (0,0,0,0,0,0), (0,0,0,0,0,0), (0,0,0,0,0,0) )`

`__init__`(*(object)arg1*) → None

___init___((object)arg1, (Matrix6c)other) -> None

___init___((object)arg1, (Vector6c)diag) -> object

___init___((object)arg1, (Matrix3c)ul, (Matrix3c)ur, (Matrix3c)ll, (Matrix3c)lr) -> object

___init___((object)arg1, (Vector6c)l0, (Vector6c)l1, (Vector6c)l2, (Vector6c)l3, (Vector6c)l4, (Vector6c)l5 [, (bool)cols=False]) -> object

`col`(*(Matrix6c)arg1*, *(int)col*) → Vector6c :

Return column as vector.

`cols`(*(Matrix6c)arg1*) → int :

Number of columns.

`determinant`(*(Matrix6c)arg1*) → complex :

Return matrix determinant.

`diagonal`(*(Matrix6c)arg1*) → Vector6c :

Return diagonal as vector.

`inverse`(*(Matrix6c)arg1*) → Matrix6c :

Return inverted matrix.

`isApprox`(*(Matrix6c)arg1*, *(Matrix6c)other*[, *(float)prec=1e-12*]) → bool :

Approximate comparison with precision *prec*.

`ll`(*(Matrix6c)arg1*) → Matrix3c :

Return lower-left 3x3 block

`lr`(*(Matrix6c)arg1*) → Matrix3c :

Return lower-right 3x3 block

`maxAbsCoeff`(*(Matrix6c)arg1*) → float :

Maximum absolute value over all elements.

`mean`(*(Matrix6c)arg1*) → complex :

Mean value over all elements.

`norm`(*(Matrix6c)arg1*) → float :

Euclidean norm.

`normalize`(*(Matrix6c)arg1*) → None :

Normalize this object in-place.

`normalized`(*(Matrix6c)arg1*) → Matrix6c :

Return normalized copy of this object

`prod`(*(Matrix6c)arg1*) → complex :

Product of all elements.

`pruned`(*(Matrix6c)arg1*[, *(float)absTol=1e-06*]) → Matrix6c :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

`row`(*(Matrix6c)arg1*, *(int)row*) → Vector6c :

Return row as vector.

`rows`(*(Matrix6c)arg1*) → int :

Number of rows.

`squaredNorm`(*(Matrix6c)arg1*) → float :

Square of the Euclidean norm.

`sum`(*(Matrix6c)arg1*) → complex :

Sum of all elements.

`trace`(*(Matrix6c)arg1*) → complex :

Return sum of diagonal elements.

`transpose`(*(Matrix6c)arg1*) → Matrix6c :

Return transposed matrix.

`ul`(*(Matrix6c)arg1*) → Matrix3c :

Return upper-left 3x3 block

`ur`(*(Matrix6c)arg1*) → Matrix3c :

Return upper-right 3x3 block

`class MatrixX`

XxX (dynamic-sized) float matrix. Constructed from list of rows (as VectorX).

Supported operations (`m` is a MatrixX, `f` if a float/int, `v` is a VectorX): `-m`, `m+m`, `m+=m`, `m-m`, `m-=m`, `m*f`, `f*m`, `m*=f`, `m/f`, `m/=f`, `m*m`, `m*=m`, `m*v`, `v*m`, `m==m`, `m!=m`.

`static Identity`(*(int)arg1*, *(int)rank*) → MatrixX :

Create identity matrix with given rank (square).

`static Ones`(*(int)rows*, *(int)cols*) → MatrixX :

Create matrix of given dimensions where all elements are set to 1.

`static Random`(*(int)rows*, *(int)cols*) → MatrixX :

Create matrix with given dimensions where all elements are set to number between 0 and 1 (uniformly-distributed).

`static Zero`(*(int)rows*, *(int)cols*) → MatrixX :

Create zero matrix of given dimensions

__init__(*(object)arg1*) → None

___init___((object)arg1, (MatrixX)other) -> None

___init___((object)arg1, (VectorX)diag) -> object

___init___((object)arg1 [, (VectorX)r0=VectorX() [, (VectorX)r1=VectorX() [, (VectorX)r2=VectorX() [, (VectorX)r3=VectorX() [, (VectorX)r4=VectorX() [, (VectorX)r5=VectorX() [, (VectorX)r6=VectorX() [, (VectorX)r7=VectorX() [, (VectorX)r8=VectorX() [, (VectorX)r9=VectorX() [, (bool)cols=False]]]]]]]]]]) -> object

___init___((object)arg1, (object)rows [, (bool)cols=False]) -> object

col(*(MatrixX)arg1*, *(int)col*) → VectorX :

Return column as vector.

cols(*(MatrixX)arg1*) → int :

Number of columns.

computeUnitaryPositive(*(MatrixX)arg1*) → tuple :

Compute polar decomposition (unitary matrix U and positive semi-definite symmetric matrix P such that self=U*P).

determinant(*(MatrixX)arg1*) → float :

Return matrix determinant.

diagonal(*(MatrixX)arg1*) → VectorX :

Return diagonal as vector.

inverse(*(MatrixX)arg1*) → MatrixX :

Return inverted matrix.

isApprox(*(MatrixX)arg1*, *(MatrixX)other*[, *(float)prec=1e-12*]) → bool :

Approximate comparison with precision *prec*.

jacobiSVD(*(MatrixX)arg1*) → tuple :

Compute SVD decomposition of square matrix, retuns (U,S,V) such that self=U*S*V.transpose()

maxAbsCoeff(*(MatrixX)arg1*) → float :

Maximum absolute value over all elements.

maxCoeff(*(MatrixX)arg1*) → float :

Maximum value over all elements.

mean(*(MatrixX)arg1*) → float :

Mean value over all elements.

minCoeff(*(MatrixX)arg1*) → float :

Minimum value over all elements.

norm(*(MatrixX)arg1*) → float :

Euclidean norm.

normalize(*(MatrixX)arg1*) → None :

Normalize this object in-place.

normalized(*(MatrixX)arg1*) → MatrixX :

Return normalized copy of this object

polarDecomposition(*(MatrixX)arg1*) → tuple :

Alias for *computeUnitaryPositive*.

prod(*(MatrixX)arg1*) → float :

Product of all elements.

pruned(*(MatrixX)arg1*[, *(float)absTol=1e-06*]) → MatrixX :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

resize(*(MatrixX)arg1*, *(int)rows*, *(int)cols*) → None :

Change size of the matrix, keep values of elements which exist in the new matrix

row(*(MatrixX)arg1*, *(int)row*) → VectorX :

Return row as vector.

rows(*(MatrixX)arg1*) → int :

Number of rows.

selfAdjointEigenDecomposition(*(MatrixX)arg1*) → tuple :

Compute eigen (spectral) decomposition of symmetric matrix, returns (eigVecs,eigVals). eigVecs is orthogonal Matrix3 with columns ar normalized eigenvectors, eigVals is Vector3 with corresponding eigenvalues. self=eigVecs*diag(eigVals)*eigVecs.transpose().

spectralDecomposition(*(MatrixX)arg1*) → tuple :

Alias for *selfAdjointEigenDecomposition*.

squaredNorm(*(MatrixX)arg1*) → float :

Square of the Euclidean norm.

sum(*(MatrixX)arg1*) → float :

Sum of all elements.

svd(*(MatrixX)arg1*) → tuple :

Alias for *jacobiSVD*.

trace(*(MatrixX)arg1*) → float :

Return sum of diagonal elements.

transpose(*(MatrixX)arg1*) → MatrixX :

Return transposed matrix.

class MatrixXc

/TODO/

static Identity(*(int)arg1*, *(int)rank*) → MatrixXc :

Create identity matrix with given rank (square).

static Ones(*(int)rows*, *(int)cols*) → MatrixXc :

Create matrix of given dimensions where all elements are set to 1.

static Random(*(int)rows*, *(int)cols*) → MatrixXc :

Create matrix with given dimensions where all elements are set to number between 0 and 1 (uniformly-distributed).

static Zero(*(int)rows*, *(int)cols*) → MatrixXc :

Create zero matrix of given dimensions

__init__(*(object)arg1*) → None

___init___((object)arg1, (MatrixXc)other) -> None

___init___((object)arg1, (VectorXc)diag) -> object

___init___((object)arg1 [, (VectorXc)r0=VectorXc() [, (VectorXc)r1=VectorXc() [, (VectorXc)r2=VectorXc() [, (VectorXc)r3=VectorXc() [, (VectorXc)r4=VectorXc() [, (VectorXc)r5=VectorXc() [, (VectorXc)r6=VectorXc() [, (VectorXc)r7=VectorXc() [, (VectorXc)r8=VectorXc() [, (VectorXc)r9=VectorXc() [, (bool)cols=False]]]]]]]]]]) -> object

___init___((object)arg1, (object)rows [, (bool)cols=False]) -> object

`col`(*(MatrixXc)arg1*, *(int)col*) → VectorXc :
Return column as vector.

`cols`(*(MatrixXc)arg1*) → int :
Number of columns.

`determinant`(*(MatrixXc)arg1*) → complex :
Return matrix determinant.

`diagonal`(*(MatrixXc)arg1*) → VectorXc :
Return diagonal as vector.

`inverse`(*(MatrixXc)arg1*) → MatrixXc :
Return inverted matrix.

`isApprox`(*(MatrixXc)arg1*, *(MatrixXc)other*[, *(float)prec=1e-12*]) → bool :
Approximate comparison with precision *prec*.

`maxAbsCoeff`(*(MatrixXc)arg1*) → float :
Maximum absolute value over all elements.

`mean`(*(MatrixXc)arg1*) → complex :
Mean value over all elements.

`norm`(*(MatrixXc)arg1*) → float :
Euclidean norm.

`normalize`(*(MatrixXc)arg1*) → None :
Normalize this object in-place.

`normalized`(*(MatrixXc)arg1*) → MatrixXc :
Return normalized copy of this object

`prod`(*(MatrixXc)arg1*) → complex :
Product of all elements.

`pruned`(*(MatrixXc)arg1*[, *(float)absTol=1e-06*]) → MatrixXc :
Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

`resize`(*(MatrixXc)arg1*, *(int)rows*, *(int)cols*) → None :
Change size of the matrix, keep values of elements which exist in the new matrix

`row`(*(MatrixXc)arg1*, *(int)row*) → VectorXc :
Return row as vector.

`rows`(*(MatrixXc)arg1*) → int :
Number of rows.

`squaredNorm`(*(MatrixXc)arg1*) → float :
Square of the Euclidean norm.

`sum`(*(MatrixXc)arg1*) → complex :
Sum of all elements.

`trace`(*(MatrixXc)arg1*) → complex :
Return sum of diagonal elements.

`transpose`(*(MatrixXc)arg1*) → MatrixXc :
Return transposed matrix.

`class Quaternion`

Quaternion representing rotation.

Supported operations (`q` is a Quaternion, `v` is a Vector3): `q*q` (rotation composition), `q*=q`, `q*v` (rotating `v` by `q`), `q==q`, `q!=q`.

Static attributes: `Identity`.

Note

Quaternion is represented as axis-angle when printed (e.g. `Identity` is `Quaternion((1,0,0),0)`, and can also be constructed from the axis-angle representation. This is however different from the data stored inside, which can be accessed by indices `[0]` (`x`), `[1]` (`y`), `[2]` (`z`), `[3]` (w). To obtain axis-angle programatically, use *`Quaternion.toAxisAngle`* which returns the tuple.

`Identity = Quaternion((1,0,0),0)`

`Rotate`(*(Quaternion)arg1*, *(Vector3)v*) → *Vector3*

`__init__`(*(object)arg1*) → None

___init___((object)arg1, (Vector3)axis, (object)angle) -> object

___init___((object)arg1, (Vector3)axis, (float)angle) -> object

___init___((object)arg1, (object)angle, (Vector3)axis) -> object

___init___((object)arg1, (float)angle, (Vector3)axis) -> object

___init___((object)arg1, (tuple)axis, (str)angle) -> object

___init___((object)arg1, (tuple)tuple) -> object

___init___((object)arg1, (Vector3)u, (Vector3)v) -> object

___init___((object)arg1, (str)str1, (str)str2, (str)str3, (str)str4) -> object

___init___((object)arg1, (float)w, (float)x, (float)y, (float)z) -> None :
Initialize from coefficients.

Note

The order of coefficients is w, x, y, z. The [] operator numbers them differently, 0...4 for x y z w!

___init___((object)arg1, (Matrix3)rotMatrix) -> None

___init___((object)arg1, (Quaternion)other) -> None

`angularDistance`(*(Quaternion)arg1*, *(Quaternion)arg2*) → float

`conjugate`(*(Quaternion)arg1*) → *Quaternion*

`inverse`(*(Quaternion)arg1*) → *Quaternion*

`norm`(*(Quaternion)arg1*) → float

`normalize`(*(Quaternion)arg1*) → None

`normalized`(*(Quaternion)arg1*) → *Quaternion*

`setFromTwoVectors`(*(Quaternion)arg1*, *(Vector3)u*, *(Vector3)v*) → None

`slerp`(*(Quaternion)arg1*, *(float)t*, *(Quaternion)other*) → *Quaternion*

toAngleAxis(*(Quaternion)arg1*) → tuple

toAxisAngle(*(Quaternion)arg1*) → tuple

toRotationMatrix(*(Quaternion)arg1*) → *Matrix3*

toRotationVector(*(Quaternion)arg1*) → *Vector3*

Real

alias of `float`

class Vector2

3-dimensional float vector.

Supported operations (`f` if a float/int, `v` is a Vector3): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`.

Implicit conversion from sequence (list, tuple, ...) of 2 floats.

Static attributes: `Zero`, `Ones`, `UnitX`, `UnitY`.

Identity = Vector2(1,0)

Ones = Vector2(1,1)

static Random() → Vector2 :

Return an object where all elements are randomly set to values between 0 and 1.

static Unit(*(int)arg1*) → *Vector2*

UnitX = Vector2(1,0)

UnitY = Vector2(0,1)

Zero = Vector2(0,0)

__init__(*(object)arg1*) → None

___init___((object)arg1, (Vector2)other) -> None

___init___((object)arg1, (str)str1, (str)str2) -> object

___init___((object)arg1, (float)x, (float)y) -> None

asDiagonal(*(Vector2)arg1*) → object :

Return diagonal matrix with this vector on the diagonal.

cols(*(Vector2)arg1*) → int :

Number of columns.

dot(*(Vector2)arg1*, *(Vector2)other*) → float :

Dot product with *other*.

isApprox(*(Vector2)arg1*, *(Vector2)other*[, *(float)prec=1e-12*]) → bool :

Approximate comparison with precision *prec*.

maxAbsCoeff(*(Vector2)arg1*) → float :

Maximum absolute value over all elements.

maxCoeff(*(Vector2)arg1*) → float :

Maximum value over all elements.

mean(*(Vector2)arg1*) → float :

Mean value over all elements.

minCoeff(*(Vector2)arg1*) → float :

Minimum value over all elements.

norm(*(Vector2)arg1*) → float :
Euclidean norm.

normalize(*(Vector2)arg1*) → None :
Normalize this object in-place.

normalized(*(Vector2)arg1*) → Vector2 :
Return normalized copy of this object

outer(*(Vector2)arg1*, *(Vector2)other*) → object :
Outer product with *other*.

prod(*(Vector2)arg1*) → float :
Product of all elements.

pruned(*(Vector2)arg1*[, *(float)absTol=1e-06*]) → Vector2 :
Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

rows(*(Vector2)arg1*) → int :
Number of rows.

squaredNorm(*(Vector2)arg1*) → float :
Square of the Euclidean norm.

sum(*(Vector2)arg1*) → float :
Sum of all elements.

class Vector2c
/TODO/

Identity = Vector2c(1,0)

Ones = Vector2c(1,1)

static Random() → Vector2c :
Return an object where all elements are randomly set to values between 0 and 1.

static Unit(*(int)arg1*) → *Vector2c*

UnitX = Vector2c(1,0)

UnitY = Vector2c(0,1)

Zero = Vector2c(0,0)

__init__(*(object)arg1*) → None
___init___((object)arg1, (Vector2c)other) -> None
___init___((object)arg1, (str)str1, (str)str2) -> object
___init___((object)arg1, (complex)x, (complex)y) -> None

asDiagonal(*(Vector2c)arg1*) → object :
Return diagonal matrix with this vector on the diagonal.

cols(*(Vector2c)arg1*) → int :
Number of columns.

dot(*(Vector2c)arg1*, *(Vector2c)other*) → complex :
Dot product with *other*.

isApprox(*(Vector2c)arg1*, *(Vector2c)other*[, *(float)prec=1e-12*]) → bool :
Approximate comparison with precision *prec*.

maxAbsCoeff(*(Vector2c)arg1*) → float :
Maximum absolute value over all elements.

mean(*(Vector2c)arg1*) → complex :
Mean value over all elements.

norm(*(Vector2c)arg1*) → float :
Euclidean norm.

normalize(*(Vector2c)arg1*) → None :
Normalize this object in-place.

normalized(*(Vector2c)arg1*) → Vector2c :
Return normalized copy of this object

outer(*(Vector2c)arg1*, *(Vector2c)other*) → object :
Outer product with *other*.

prod(*(Vector2c)arg1*) → complex :
Product of all elements.

pruned(*(Vector2c)arg1*[, *(float)absTol=1e-06*]) → Vector2c :
Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

rows(*(Vector2c)arg1*) → int :
Number of rows.

squaredNorm(*(Vector2c)arg1*) → float :
Square of the Euclidean norm.

sum(*(Vector2c)arg1*) → complex :
Sum of all elements.

class Vector2i
2-dimensional integer vector.

Supported operations (`i` if an int, `v` is a Vector2i): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*i`, `i*v`, `v*=i`, `v==v`, `v!=v`.

Implicit conversion from sequence (list, tuple, ...) of 2 integers.

Static attributes: `Zero`, `Ones`, `UnitX`, `UnitY`.

Identity = Vector2i(1,0)

Ones = Vector2i(1,1)

static Random() → Vector2i :
Return an object where all elements are randomly set to values between 0 and 1.

static Unit(*(int)arg1*) → *Vector2i*

UnitX = Vector2i(1,0)

UnitY = Vector2i(0,1)

Zero = Vector2i(0,0)

__init__(*(object)arg1*) → None
___init___((object)arg1, (Vector2i)other) -> None

___init___((object)arg1, (str)str1, (str)str2) -> object

___init___((object)arg1, (int)x, (int)y) -> None

`asDiagonal`(*(Vector2i)arg1*) → object :

Return diagonal matrix with this vector on the diagonal.

`cols`(*(Vector2i)arg1*) → int :

Number of columns.

`dot`(*(Vector2i)arg1*, *(Vector2i)other*) → int :

Dot product with *other*.

`isApprox`(*(Vector2i)arg1*, *(Vector2i)other*[, *(int)prec=0*]) → bool :

Approximate comparison with precision *prec*.

`maxAbsCoeff`(*(Vector2i)arg1*) → int :

Maximum absolute value over all elements.

`maxCoeff`(*(Vector2i)arg1*) → int :

Maximum value over all elements.

`mean`(*(Vector2i)arg1*) → int :

Mean value over all elements.

`minCoeff`(*(Vector2i)arg1*) → int :

Minimum value over all elements.

`outer`(*(Vector2i)arg1*, *(Vector2i)other*) → object :

Outer product with *other*.

`prod`(*(Vector2i)arg1*) → int :

Product of all elements.

`rows`(*(Vector2i)arg1*) → int :

Number of rows.

`sum`(*(Vector2i)arg1*) → int :

Sum of all elements.

`class Vector3`

3-dimensional float vector.

Supported operations (`f` if a float/int, `v` is a Vector3): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`, plus operations with `Matrix3` and `Quaternion`.

Implicit conversion from sequence (list, tuple, ...) of 3 floats.

Static attributes: `Zero`, `Ones`, `UnitX`, `UnitY`, `UnitZ`.

`Identity = Vector3(1,0,0)`

`Ones = Vector3(1,1,1)`

`static Random()` → Vector3 :

Return an object where all elements are randomly set to values between 0 and 1.

`static Unit`(*(int)arg1*) → *Vector3*

`UnitX = Vector3(1,0,0)`

`UnitY = Vector3(0,1,0)`

`UnitZ = Vector3(0,0,1)`

`Zero = Vector3(0,0,0)`

__init__(*(object)arg1*) → None

___init___((object)arg1, (Vector3)other) -> None

___init___((object)arg1, (str)str1, (str)str2, (str)str3) -> object

___init___((object)arg1 [, (float)x=0.0 [, (float)y=0.0 [, (float)z=0.0]]]) -> None

asDiagonal(*(Vector3)arg1*) → Matrix3 :

Return diagonal matrix with this vector on the diagonal.

cols(*(Vector3)arg1*) → int :

Number of columns.

cross(*(Vector3)arg1*, *(Vector3)arg2*) → *Vector3*

dot(*(Vector3)arg1*, *(Vector3)other*) → float :

Dot product with *other*.

isApprox(*(Vector3)arg1*, *(Vector3)other*[, *(float)prec=1e-12*]) → bool :

Approximate comparison with precision *prec*.

maxAbsCoeff(*(Vector3)arg1*) → float :

Maximum absolute value over all elements.

maxCoeff(*(Vector3)arg1*) → float :

Maximum value over all elements.

mean(*(Vector3)arg1*) → float :

Mean value over all elements.

minCoeff(*(Vector3)arg1*) → float :

Minimum value over all elements.

norm(*(Vector3)arg1*) → float :

Euclidean norm.

normalize(*(Vector3)arg1*) → None :

Normalize this object in-place.

normalized(*(Vector3)arg1*) → Vector3 :

Return normalized copy of this object

outer(*(Vector3)arg1*, *(Vector3)other*) → Matrix3 :

Outer product with *other*.

prod(*(Vector3)arg1*) → float :

Product of all elements.

pruned(*(Vector3)arg1*[, *(float)absTol=1e-06*]) → Vector3 :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

rows(*(Vector3)arg1*) → int :

Number of rows.

squaredNorm(*(Vector3)arg1*) → float :

Square of the Euclidean norm.

sum(*(Vector3)arg1*) → float :

Sum of all elements.

xy(*(Vector3)arg1*) → *Vector2*

xz(*(Vector3)arg1*) → *Vector2*

yx(*(Vector3)arg1*) → *Vector2*

yz(*(Vector3)arg1*) → *Vector2*

zx(*(Vector3)arg1*) → *Vector2*

zy(*(Vector3)arg1*) → *Vector2*

class Vector3c

/TODO/

Identity = Vector3c(1,0,0)

Ones = Vector3c(1,1,1)

static Random() → Vector3c :

Return an object where all elements are randomly set to values between 0 and 1.

static Unit(*(int)arg1*) → *Vector3c*

UnitX = Vector3c(1,0,0)

UnitY = Vector3c(0,1,0)

UnitZ = Vector3c(0,0,1)

Zero = Vector3c(0,0,0)

__init__(*(object)arg1*) → None

___init___((object)arg1, (Vector3c)other) -> None

___init___((object)arg1, (str)str1, (str)str2, (str)str3) -> object

___init___((object)arg1 [, (complex)x=0j [, (complex)y=0j [, (complex)z=0j]]]) -> None

asDiagonal(*(Vector3c)arg1*) → Matrix3c :

Return diagonal matrix with this vector on the diagonal.

cols(*(Vector3c)arg1*) → int :

Number of columns.

cross(*(Vector3c)arg1*, *(Vector3c)arg2*) → *Vector3c*

dot(*(Vector3c)arg1*, *(Vector3c)other*) → complex :

Dot product with *other*.

isApprox(*(Vector3c)arg1*, *(Vector3c)other*[, *(float)prec=1e-12*]) → bool :

Approximate comparison with precision *prec*.

maxAbsCoeff(*(Vector3c)arg1*) → float :

Maximum absolute value over all elements.

mean(*(Vector3c)arg1*) → complex :

Mean value over all elements.

norm(*(Vector3c)arg1*) → float :

Euclidean norm.

normalize(*(Vector3c)arg1*) → None :

Normalize this object in-place.

normalized(*(Vector3c)arg1*) → Vector3c :

Return normalized copy of this object

outer(*(Vector3c)arg1*, *(Vector3c)other*) → Matrix3c :
Outer product with *other*.

prod(*(Vector3c)arg1*) → complex :
Product of all elements.

pruned(*(Vector3c)arg1*[, *(float)absTol=1e-06*]) → Vector3c :
Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

rows(*(Vector3c)arg1*) → int :
Number of rows.

squaredNorm(*(Vector3c)arg1*) → float :
Square of the Euclidean norm.

sum(*(Vector3c)arg1*) → complex :
Sum of all elements.

xy(*(Vector3c)arg1*) → *Vector2c*

xz(*(Vector3c)arg1*) → *Vector2c*

yx(*(Vector3c)arg1*) → *Vector2c*

yz(*(Vector3c)arg1*) → *Vector2c*

zx(*(Vector3c)arg1*) → *Vector2c*

zy(*(Vector3c)arg1*) → *Vector2c*

class Vector3i
3-dimensional integer vector.

Supported operations (`i` if an int, `v` is a Vector3i): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*i`, `i*v`, `v*=i`, `v==v`, `v!=v`.

Implicit conversion from sequence (list, tuple, ...) of 3 integers.

Static attributes: **Zero**, **Ones**, **UnitX**, **UnitY**, **UnitZ**.

Identity = Vector3i(1,0,0)

Ones = Vector3i(1,1,1)

static Random() → Vector3i :
Return an object where all elements are randomly set to values between 0 and 1.

static Unit(*(int)arg1*) → *Vector3i*

UnitX = Vector3i(1,0,0)

UnitY = Vector3i(0,1,0)

UnitZ = Vector3i(0,0,1)

Zero = Vector3i(0,0,0)

__init__(*(object)arg1*) → None
___init___((object)arg1, (Vector3i)other) -> None

___init___((object)arg1, (str)str1, (str)str2, (str)str3) -> object

___init___((object)arg1 [, (int)x=0 [, (int)y=0 [, (int)z=0]]]) -> None

asDiagonal(*(Vector3i)arg1*) → object :
Return diagonal matrix with this vector on the diagonal.

`cols`(*(Vector3i)arg1*) → int :
Number of columns.

`cross`(*(Vector3i)arg1*, *(Vector3i)arg2*) → *Vector3i*

`dot`(*(Vector3i)arg1*, *(Vector3i)other*) → int :
Dot product with *other*.

`isApprox`(*(Vector3i)arg1*, *(Vector3i)other*[, *(int)prec=0*]) → bool :
Approximate comparison with precision *prec*.

`maxAbsCoeff`(*(Vector3i)arg1*) → int :
Maximum absolute value over all elements.

`maxCoeff`(*(Vector3i)arg1*) → int :
Maximum value over all elements.

`mean`(*(Vector3i)arg1*) → int :
Mean value over all elements.

`minCoeff`(*(Vector3i)arg1*) → int :
Minimum value over all elements.

`outer`(*(Vector3i)arg1*, *(Vector3i)other*) → object :
Outer product with *other*.

`prod`(*(Vector3i)arg1*) → int :
Product of all elements.

`rows`(*(Vector3i)arg1*) → int :
Number of rows.

`sum`(*(Vector3i)arg1*) → int :
Sum of all elements.

`xy`(*(Vector3i)arg1*) → *Vector2i*

`xz`(*(Vector3i)arg1*) → *Vector2i*

`yx`(*(Vector3i)arg1*) → *Vector2i*

`yz`(*(Vector3i)arg1*) → *Vector2i*

`zx`(*(Vector3i)arg1*) → *Vector2i*

`zy`(*(Vector3i)arg1*) → *Vector2i*

`class Vector4`
4-dimensional float vector.

Supported operations (`f` if a float/int, `v` is a Vector3): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`.

Implicit conversion from sequence (list, tuple, ...) of 4 floats.

Static attributes: `Zero`, `Ones`.

`Identity = Vector4(1,0,0, 0)`

`Ones = Vector4(1,1,1, 1)`

`static Random()` → Vector4 :
Return an object where all elements are randomly set to values between 0 and 1.

static Unit(*(int)arg1*) → *Vector4*

Zero = Vector4(0,0,0, 0)

__init__(*(object)arg1*) → None

___init___((object)arg1, (Vector4)other) -> None

___init___((object)arg1, (str)str1, (str)str2, (str)str3, (str)str4) -> object

___init___((object)arg1, (float)v0, (float)v1, (float)v2, (float)v3) -> None

asDiagonal(*(Vector4)arg1*) → object :

Return diagonal matrix with this vector on the diagonal.

cols(*(Vector4)arg1*) → int :

Number of columns.

dot(*(Vector4)arg1*, *(Vector4)other*) → float :

Dot product with *other*.

isApprox(*(Vector4)arg1*, *(Vector4)other*[, *(float)prec=1e-12*]) → bool :

Approximate comparison with precision *prec*.

maxAbsCoeff(*(Vector4)arg1*) → float :

Maximum absolute value over all elements.

maxCoeff(*(Vector4)arg1*) → float :

Maximum value over all elements.

mean(*(Vector4)arg1*) → float :

Mean value over all elements.

minCoeff(*(Vector4)arg1*) → float :

Minimum value over all elements.

norm(*(Vector4)arg1*) → float :

Euclidean norm.

normalize(*(Vector4)arg1*) → None :

Normalize this object in-place.

normalized(*(Vector4)arg1*) → Vector4 :

Return normalized copy of this object

outer(*(Vector4)arg1*, *(Vector4)other*) → object :

Outer product with *other*.

prod(*(Vector4)arg1*) → float :

Product of all elements.

pruned(*(Vector4)arg1*[, *(float)absTol=1e-06*]) → Vector4 :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

rows(*(Vector4)arg1*) → int :

Number of rows.

squaredNorm(*(Vector4)arg1*) → float :

Square of the Euclidean norm.

sum(*(Vector4)arg1*) → float :

Sum of all elements.

`class Vector6`

6-dimensional float vector.

Supported operations (`f` if a float/int, `v` is a Vector6): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`.

Implicit conversion from sequence (list, tuple, ...) of 6 floats.

Static attributes: `Zero`, `Ones`.

`Identity = Vector6(1,0,0, 0,0,0)`

`Ones = Vector6(1,1,1, 1,1,1)`

`static Random()` → Vector6 :

Return an object where all elements are randomly set to values between 0 and 1.

`static Unit(`*(int)arg1*`)` → *Vector6*

`Zero = Vector6(0,0,0, 0,0,0)`

`__init__(`*(object)arg1*`)` → None

___init___((object)arg1, (Vector6)other) -> None

___init___((object)arg1, (float)v0, (float)v1, (float)v2, (float)v3, (float)v4, (float)v5) -> object

___init___((object)arg1, (Vector3)head, (Vector3)tail) -> object

`asDiagonal(`*(Vector6)arg1*`)` → Matrix6 :

Return diagonal matrix with this vector on the diagonal.

`cols(`*(Vector6)arg1*`)` → int :

Number of columns.

`dot(`*(Vector6)arg1*, *(Vector6)other*`)` → float :

Dot product with *other*.

`head(`*(Vector6)arg1*`)` → *Vector3*

`isApprox(`*(Vector6)arg1*, *(Vector6)other*[, *(float)prec=1e-12*]`)` → bool :

Approximate comparison with precision *prec*.

`maxAbsCoeff(`*(Vector6)arg1*`)` → float :

Maximum absolute value over all elements.

`maxCoeff(`*(Vector6)arg1*`)` → float :

Maximum value over all elements.

`mean(`*(Vector6)arg1*`)` → float :

Mean value over all elements.

`minCoeff(`*(Vector6)arg1*`)` → float :

Minimum value over all elements.

`norm(`*(Vector6)arg1*`)` → float :

Euclidean norm.

`normalize(`*(Vector6)arg1*`)` → None :

Normalize this object in-place.

`normalized(`*(Vector6)arg1*`)` → Vector6 :

Return normalized copy of this object

outer(*(Vector6)arg1*, *(Vector6)other*) → Matrix6 :
Outer product with *other*.

prod(*(Vector6)arg1*) → float :
Product of all elements.

pruned(*(Vector6)arg1*[, *(float)absTol=1e-06*]) → Vector6 :
Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

rows(*(Vector6)arg1*) → int :
Number of rows.

squaredNorm(*(Vector6)arg1*) → float :
Square of the Euclidean norm.

sum(*(Vector6)arg1*) → float :
Sum of all elements.

tail(*(Vector6)arg1*) → *Vector3*

class Vector6c
/TODO/

Identity = Vector6c(1,0,0, 0,0,0)

Ones = Vector6c(1,1,1, 1,1,1)

static Random() → Vector6c :
Return an object where all elements are randomly set to values between 0 and 1.

static Unit(*(int)arg1*) → *Vector6c*

Zero = Vector6c(0,0,0, 0,0,0)

__init__(*(object)arg1*) → None
___init___((object)arg1, (Vector6c)other) -> None

___init___((object)arg1, (complex)v0, (complex)v1, (complex)v2, (complex)v3, (complex)v4, (complex)v5) -> object

___init___((object)arg1, (Vector3c)head, (Vector3c)tail) -> object

asDiagonal(*(Vector6c)arg1*) → Matrix6c :
Return diagonal matrix with this vector on the diagonal.

cols(*(Vector6c)arg1*) → int :
Number of columns.

dot(*(Vector6c)arg1*, *(Vector6c)other*) → complex :
Dot product with *other*.

head(*(Vector6c)arg1*) → *Vector3c*

isApprox(*(Vector6c)arg1*, *(Vector6c)other*[, *(float)prec=1e-12*]) → bool :
Approximate comparison with precision *prec*.

maxAbsCoeff(*(Vector6c)arg1*) → float :
Maximum absolute value over all elements.

mean(*(Vector6c)arg1*) → complex :
Mean value over all elements.

norm(*(Vector6c)arg1*) → float :
Euclidean norm.

normalize(*(Vector6c)arg1*) → None :
Normalize this object in-place.

normalized(*(Vector6c)arg1*) → Vector6c :
Return normalized copy of this object

outer(*(Vector6c)arg1*, *(Vector6c)other*) → Matrix6c :
Outer product with *other*.

prod(*(Vector6c)arg1*) → complex :
Product of all elements.

pruned(*(Vector6c)arg1*[, *(float)absTol=1e-06*]) → Vector6c :
Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

rows(*(Vector6c)arg1*) → int :
Number of rows.

squaredNorm(*(Vector6c)arg1*) → float :
Square of the Euclidean norm.

sum(*(Vector6c)arg1*) → complex :
Sum of all elements.

tail(*(Vector6c)arg1*) → *Vector3c*

class Vector6i
6-dimensional float vector.

Supported operations (`f` if a float/int, `v` is a Vector6): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`.

Implicit conversion from sequence (list, tuple, ...) of 6 ints.

Static attributes: `Zero`, `Ones`.

Identity = Vector6i(1,0,0, 0,0,0)

Ones = Vector6i(1,1,1, 1,1,1)

static Random() → Vector6i :
Return an object where all elements are randomly set to values between 0 and 1.

static Unit(*(int)arg1*) → *Vector6i*

Zero = Vector6i(0,0,0, 0,0,0)

__init__(*(object)arg1*) → None
___init___((object)arg1, (Vector6i)other) -> None

___init___((object)arg1, (int)v0, (int)v1, (int)v2, (int)v3, (int)v4, (int)v5) -> object

___init___((object)arg1, (Vector3i)head, (Vector3i)tail) -> object

asDiagonal(*(Vector6i)arg1*) → object :
Return diagonal matrix with this vector on the diagonal.

cols(*(Vector6i)arg1*) → int :
Number of columns.

dot(*(Vector6i)arg1*, *(Vector6i)other*) → int :
Dot product with *other*.

head(*(Vector6i)arg1*) → *Vector3i*

isApprox(*(Vector6i)arg1*, *(Vector6i)other*[, *(int)prec=0*]) → bool :
Approximate comparison with precision *prec*.

maxAbsCoeff(*(Vector6i)arg1*) → int :
Maximum absolute value over all elements.

maxCoeff(*(Vector6i)arg1*) → int :
Maximum value over all elements.

mean(*(Vector6i)arg1*) → int :
Mean value over all elements.

minCoeff(*(Vector6i)arg1*) → int :
Minimum value over all elements.

outer(*(Vector6i)arg1*, *(Vector6i)other*) → object :
Outer product with *other*.

prod(*(Vector6i)arg1*) → int :
Product of all elements.

rows(*(Vector6i)arg1*) → int :
Number of rows.

sum(*(Vector6i)arg1*) → int :
Sum of all elements.

tail(*(Vector6i)arg1*) → *Vector3i*

class VectorX
Dynamic-sized float vector.

Supported operations (`f` if a float/int, `v` is a VectorX): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`.

Implicit conversion from sequence (list, tuple, ...) of X floats.

static Ones(*(int)arg1*) → *VectorX*

static Random(*(int)len*) → VectorX :
Return vector of given length with all elements set to values between 0 and 1 randomly.

static Unit(*(int)arg1*, *(int)arg2*) → *VectorX*

static Zero(*(int)arg1*) → *VectorX*

__init__(*(object)arg1*) → None
___init___((object)arg1, (VectorX)other) -> None

___init___((object)arg1, (object)vv) -> object

asDiagonal(*(VectorX)arg1*) → MatrixX :
Return diagonal matrix with this vector on the diagonal.

cols(*(VectorX)arg1*) → int :
Number of columns.

dot(*(VectorX)arg1*, *(VectorX)other*) → float :
Dot product with *other*.

isApprox(*(VectorX)arg1*, *(VectorX)other*[, *(float)prec=1e-12*]) → bool :
Approximate comparison with precision *prec*.

maxAbsCoeff(*(VectorX)arg1*) → float :
Maximum absolute value over all elements.

maxCoeff(*(VectorX)arg1*) → float :
Maximum value over all elements.

mean(*(VectorX)arg1*) → float :
Mean value over all elements.

minCoeff(*(VectorX)arg1*) → float :
Minimum value over all elements.

norm(*(VectorX)arg1*) → float :
Euclidean norm.

normalize(*(VectorX)arg1*) → None :
Normalize this object in-place.

normalized(*(VectorX)arg1*) → VectorX :
Return normalized copy of this object

outer(*(VectorX)arg1*, *(VectorX)other*) → MatrixX :
Outer product with *other*.

prod(*(VectorX)arg1*) → float :
Product of all elements.

pruned(*(VectorX)arg1*[, *(float)absTol=1e-06*]) → VectorX :
Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

resize(*(VectorX)arg1*, *(int)arg2*) → None

rows(*(VectorX)arg1*) → int :
Number of rows.

squaredNorm(*(VectorX)arg1*) → float :
Square of the Euclidean norm.

sum(*(VectorX)arg1*) → float :
Sum of all elements.

class VectorXc

/TODO/

static Ones(*(int)arg1*) → *VectorXc*

static Random(*(int)len*) → VectorXc :
Return vector of given length with all elements set to values between 0 and 1 randomly.

static Unit(*(int)arg1*, *(int)arg2*) → *VectorXc*

static Zero(*(int)arg1*) → *VectorXc*

__init__(*(object)arg1*) → None
___init___((object)arg1, (VectorXc)other) -> None

___init___((object)arg1, (object)vv) -> object

asDiagonal(*(VectorXc)arg1*) → MatrixXc :
Return diagonal matrix with this vector on the diagonal.

cols(*(VectorXc)arg1*) → int :
Number of columns.

dot(*(VectorXc)arg1*, *(VectorXc)other*) → complex :
Dot product with *other*.

isApprox(*(VectorXc)arg1*, *(VectorXc)other*[, *(float)prec=1e-12*]) → bool :
Approximate comparison with precision *prec*.

maxAbsCoeff(*(VectorXc)arg1*) → float :
Maximum absolute value over all elements.

mean(*(VectorXc)arg1*) → complex :
Mean value over all elements.

norm(*(VectorXc)arg1*) → float :
Euclidean norm.

normalize(*(VectorXc)arg1*) → None :
Normalize this object in-place.

normalized(*(VectorXc)arg1*) → VectorXc :
Return normalized copy of this object

outer(*(VectorXc)arg1*, *(VectorXc)other*) → MatrixXc :
Outer product with *other*.

prod(*(VectorXc)arg1*) → complex :
Product of all elements.

pruned(*(VectorXc)arg1*[, *(float)absTol=1e-06*]) → VectorXc :
Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

resize(*(VectorXc)arg1*, *(int)arg2*) → None

rows(*(VectorXc)arg1*) → int :
Number of rows.

squaredNorm(*(VectorXc)arg1*) → float :
Square of the Euclidean norm.

sum(*(VectorXc)arg1*) → complex :
Sum of all elements.

vectorize = False

class yade._minieigenHP.**HP2**

class AlignedBox2
Axis-aligned box object in 2d, defined by its minimum and maximum corners

__init__(*(object)arg1*) → None
___init___((object)arg1, (HP2.AlignedBox2)other) -> None
___init___((object)arg1, (HP2.Vector2)min, (HP2.Vector2)max) -> None

center(*(HP2.AlignedBox2)arg1*) → *HP2.Vector2*

clamp(*(HP2.AlignedBox2)arg1*, *(HP2.AlignedBox2)arg2*) → None

contains(*(HP2.AlignedBox2)arg1*, *(HP2.Vector2)arg2*) → bool
contains((HP2.AlignedBox2)arg1, (HP2.AlignedBox2)arg2) -> bool

empty(*(HP2.AlignedBox2)arg1*) → bool

extend(*(HP2.AlignedBox2)arg1*, *(HP2.Vector2)arg2*) → None
extend((HP2.AlignedBox2)arg1, (HP2.AlignedBox2)arg2) -> None

intersection(*(HP2.AlignedBox2)arg1*, *(HP2.AlignedBox2)arg2*) → *HP2.AlignedBox2*

property max
None((yade._minieigenHP.HP2.AlignedBox2)arg1) -> yade._minieigenHP.HP2.Vector2

merged(*(HP2.AlignedBox2)arg1*, *(HP2.AlignedBox2)arg2*) → *HP2.AlignedBox2*

property min
None((yade._minieigenHP.HP2.AlignedBox2)arg1) -> yade._minieigenHP.HP2.Vector2

sizes(*(HP2.AlignedBox2)arg1*) → *HP2.Vector2*

volume(*(HP2.AlignedBox2)arg1*) → *HP2.Real*

class AlignedBox3
Axis-aligned box object, defined by its minimum and maximum corners

__init__(*(object)arg1*) → None
___init___((object)arg1, (HP2.AlignedBox3)other) -> None

___init___((object)arg1, (HP2.Vector3)min, (HP2.Vector3)max) -> None

center(*(HP2.AlignedBox3)arg1*) → *HP2.Vector3*

clamp(*(HP2.AlignedBox3)arg1*, *(HP2.AlignedBox3)arg2*) → None

contains(*(HP2.AlignedBox3)arg1*, *(HP2.Vector3)arg2*) → bool
contains((HP2.AlignedBox3)arg1, (HP2.AlignedBox3)arg2) -> bool

empty(*(HP2.AlignedBox3)arg1*) → bool

extend(*(HP2.AlignedBox3)arg1*, *(HP2.Vector3)arg2*) → None
extend((HP2.AlignedBox3)arg1, (HP2.AlignedBox3)arg2) -> None

intersection(*(HP2.AlignedBox3)arg1*, *(HP2.AlignedBox3)arg2*) → *HP2.AlignedBox3*

property max
None((yade._minieigenHP.HP2.AlignedBox3)arg1) -> yade._minieigenHP.HP2.Vector3

merged(*(HP2.AlignedBox3)arg1*, *(HP2.AlignedBox3)arg2*) → *HP2.AlignedBox3*

property min
None((yade._minieigenHP.HP2.AlignedBox3)arg1) -> yade._minieigenHP.HP2.Vector3

sizes(*(HP2.AlignedBox3)arg1*) → *HP2.Vector3*

volume(*(HP2.AlignedBox3)arg1*) → *HP2.Real*

class Complex
The Complex type.

__init__(*(object)arg1*) → None
___init___((object)arg1, (object)obj) -> object

___init___((object)arg1, (complex)z) -> object

___init___((object)arg1, (float)d) -> object

___init___((object)arg1, (int)i) -> object

___init___((object)arg1, (str)str) -> object

___init___((object)arg1, (object)re, (object)im) -> object

___init___((object)arg1, (float)a, (float)b) -> object

___init___((object)arg1, (int)i, (int)j) -> object

___init___((object)arg1, (str)str1, (str)str2) -> object

property imag

None((yade._minieigenHP.HP2.Complex)arg1) -> object

levelComplexHPMethod(*(HP2.Complex)arg1*) → int

property levelHP

None((yade._minieigenHP.HP2.Complex)arg1) -> int

property real

None((yade._minieigenHP.HP2.Complex)arg1) -> object

class Matrix3

3x3 float matrix.

Supported operations (`m` is a Matrix3, `f` if a float/int, `v` is a Vector3): `-m`, `m+m`, `m+=m`, `m-m`, `m-=m`, `m*f`, `f*m`, `m*=f`, `m/f`, `m/=f`, `m*m`, `m*=m`, `m*v`, `v*m`, `m==m`, `m!=m`.

Static attributes: `Zero`, `Ones`, `Identity`.

Identity = Matrix3("1","0","0", "0","1","0", "0","0","1")

Ones = Matrix3("1","1","1", "1","1","1", "1","1","1")

static Random() → HP2.Matrix3 :

Return an object where all elements are randomly set to values between 0 and 1.

Zero = Matrix3("0","0","0", "0","0","0", "0","0","0")

__init__(*(object)arg1*) → None

___init___((object)arg1, (HP2.Quaternion)q) -> None

___init___((object)arg1, (HP2.Matrix3)other) -> None

___init___((object)arg1, (HP2.Vector3)diag) -> object

___init___((object)arg1, (HP2.Real)m00, (HP2.Real)m01, (HP2.Real)m02, (HP2.Real)m10, (HP2.Real)m11, (HP2.Real)m12, (HP2.Real)m20, (HP2.Real)m21, (HP2.Real)m22) -> object

___init___((object)arg1, (str)m00, (str)m01, (str)m02, (str)m10, (str)m11, (str)m12, (str)m20, (str)m21, (str)m22) -> object

___init___((object)arg1, (HP2.Vector3)r0, (HP2.Vector3)r1, (HP2.Vector3)r2 [, (bool)cols=False]) -> object

col(*(HP2.Matrix3)arg1*, *(int)col*) → HP2.Vector3 :

Return column as vector.

cols(*(HP2.Matrix3)arg1*) → int :

Number of columns.

computeUnitaryPositive(*(HP2.Matrix3)arg1*) → tuple :

Compute polar decomposition (unitary matrix U and positive semi-definite symmetric matrix P such that self=U*P).

determinant(*(HP2.Matrix3)arg1*) → HP2.Real :

Return matrix determinant.

diagonal(*(HP2.Matrix3)arg1*) → HP2.Vector3 :

Return diagonal as vector.

inverse(*(HP2.Matrix3)arg1*) → HP2.Matrix3 :

Return inverted matrix.

isApprox(*(HP2.Matrix3)arg1*, *(HP2.Matrix3)other*[, *(HP2.Real)prec=Real("3.84273543930596175698289618669828536 4e-31")*]) → bool :

Approximate comparison with precision *prec*.

jacobiSVD(*(HP2.Matrix3)arg1*) → tuple :

Compute SVD decomposition of square matrix, retuns (U,S,V) such that self=U*S*V.transpose()

maxAbsCoeff(*(HP2.Matrix3)arg1*) → HP2.Real :

Maximum absolute value over all elements.

maxCoeff(*(HP2.Matrix3)arg1*) → HP2.Real :

Maximum value over all elements.

mean(*(HP2.Matrix3)arg1*) → HP2.Real :

Mean value over all elements.

minCoeff(*(HP2.Matrix3)arg1*) → HP2.Real :

Minimum value over all elements.

norm(*(HP2.Matrix3)arg1*) → HP2.Real :

Euclidean norm.

normalize(*(HP2.Matrix3)arg1*) → None :

Normalize this object in-place.

normalized(*(HP2.Matrix3)arg1*) → HP2.Matrix3 :

Return normalized copy of this object

polarDecomposition(*(HP2.Matrix3)arg1*) → tuple :

Alias for *computeUnitaryPositive*.

prod(*(HP2.Matrix3)arg1*) → HP2.Real :

Product of all elements.

pruned(*(HP2.Matrix3)arg1*[, *(float)absTol=1e-06*]) → HP2.Matrix3 :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

row(*(HP2.Matrix3)arg1*, *(int)row*) → HP2.Vector3 :

Return row as vector.

rows(*(HP2.Matrix3)arg1*) → int :

Number of rows.

selfAdjointEigenDecomposition(*(HP2.Matrix3)arg1*) → tuple :

Compute eigen (spectral) decomposition of symmetric matrix, returns (eigVecs,eigVals). eigVecs is orthogonal Matrix3 with columns ar normalized eigenvectors, eigVals is Vector3 with corresponding eigenvalues. self=eigVecs*diag(eigVals)*eigVecs.transpose().

spectralDecomposition(*(HP2.Matrix3)arg1*) → tuple :

Alias for *selfAdjointEigenDecomposition*.

squaredNorm(*(HP2.Matrix3)arg1*) → HP2.Real :

Square of the Euclidean norm.

sum(*(HP2.Matrix3)arg1*) → HP2.Real :

Sum of all elements.

svd(*(HP2.Matrix3)arg1*) → tuple :

Alias for *jacobiSVD*.

`trace`(*(HP2.Matrix3)arg1*) → HP2.Real :

Return sum of diagonal elements.

`transpose`(*(HP2.Matrix3)arg1*) → HP2.Matrix3 :

Return transposed matrix.

`class Matrix3c`

/TODO/

```
Identity = Matrix3c(Complex("1","0"),Complex("0","0"),Complex("0","0"),
Complex("0","0"),Complex("1","0"),Complex("0","0"),
Complex("0","0"),Complex("0","0"),Complex("1","0"))

Ones = Matrix3c(Complex("1","0"),Complex("1","0"),Complex("1","0"),
Complex("1","0"),Complex("1","0"),Complex("1","0"),
Complex("1","0"),Complex("1","0"),Complex("1","0"))
```

`static Random`() → HP2.Matrix3c :

Return an object where all elements are randomly set to values between 0 and 1.

```
Zero = Matrix3c(Complex("0","0"),Complex("0","0"),Complex("0","0"),
Complex("0","0"),Complex("0","0"),Complex("0","0"),
Complex("0","0"),Complex("0","0"),Complex("0","0"))
```

`__init__`(*(object)arg1*) → None

___init___((object)arg1, (HP2.Matrix3c)other) -> None

___init___((object)arg1, (HP2.Vector3c)diag) -> object

___init___((object)arg1, (HP2.Complex)m00, (HP2.Complex)m01, (HP2.Complex)m02, (HP2.Complex)m10, (HP2.Complex)m11, (HP2.Complex)m12, (HP2.Complex)m20, (HP2.Complex)m21, (HP2.Complex)m22) -> object

___init___((object)arg1, (str)m00, (str)m01, (str)m02, (str)m10, (str)m11, (str)m12, (str)m20, (str)m21, (str)m22) -> object

___init___((object)arg1, (HP2.Vector3c)r0, (HP2.Vector3c)r1, (HP2.Vector3c)r2 [, (bool)cols=False]) -> object

`col`(*(HP2.Matrix3c)arg1*, *(int)col*) → HP2.Vector3c :

Return column as vector.

`cols`(*(HP2.Matrix3c)arg1*) → int :

Number of columns.

`determinant`(*(HP2.Matrix3c)arg1*) → HP2.Complex :

Return matrix determinant.

`diagonal`(*(HP2.Matrix3c)arg1*) → HP2.Vector3c :

Return diagonal as vector.

`inverse`(*(HP2.Matrix3c)arg1*) → HP2.Matrix3c :

Return inverted matrix.

`isApprox`(*(HP2.Matrix3c)arg1*, *(HP2.Matrix3c)other*[, *(HP2.Real)prec=Complex("3.842735439305961756982896186698285364e-31", "0")*]) → bool :

Approximate comparison with precision *prec*.

`maxAbsCoeff`(*(HP2.Matrix3c)arg1*) → HP2.Real :

Maximum absolute value over all elements.

`mean`(*(HP2.Matrix3c)arg1*) → HP2.Complex :

Mean value over all elements.

`norm`(*(HP2.Matrix3c)arg1*) → HP2.Real :

Euclidean norm.

`normalize`(*(HP2.Matrix3c)arg1*) → None :

Normalize this object in-place.

`normalized`(*(HP2.Matrix3c)arg1*) → HP2.Matrix3c :

Return normalized copy of this object

`prod`(*(HP2.Matrix3c)arg1*) → HP2.Complex :

Product of all elements.

`pruned`(*(HP2.Matrix3c)arg1*[, *(float)absTol=1e-06*]) → HP2.Matrix3c :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

`row`(*(HP2.Matrix3c)arg1*, *(int)row*) → HP2.Vector3c :

Return row as vector.

`rows`(*(HP2.Matrix3c)arg1*) → int :

Number of rows.

`squaredNorm`(*(HP2.Matrix3c)arg1*) → HP2.Real :

Square of the Euclidean norm.

`sum`(*(HP2.Matrix3c)arg1*) → HP2.Complex :

Sum of all elements.

`trace`(*(HP2.Matrix3c)arg1*) → HP2.Complex :

Return sum of diagonal elements.

`transpose`(*(HP2.Matrix3c)arg1*) → HP2.Matrix3c :

Return transposed matrix.

`class Matrix6`

6x6 float matrix. Constructed from 4 3x3 sub-matrices, from 6xVector6 (rows).

Supported operations (`m` is a Matrix6, `f` if a float/int, `v` is a Vector6): `-m`, `m+m`, `m+=m`, `m-m`, `m-=m`, `m*f`, `f*m`, `m*=f`, `m/f`, `m/=f`, `m*m`, `m*=m`, `m*v`, `v*m`, `m==m`, `m!=m`.

Static attributes: `Zero`, `Ones`, `Identity`.

```
Identity = Matrix6( ("1","0","0","0","0","0"), ("0","1","0","0","0","0"),
("0","0","1","0","0","0"), ("0","0","0","1","0","0"),
("0","0","0","0","1","0"), ("0","0","0","0","0","1") )

Ones = Matrix6( ("1","1","1","1","1","1"), ("1","1","1","1","1","1"),
("1","1","1","1","1","1"), ("1","1","1","1","1","1"),
("1","1","1","1","1","1"), ("1","1","1","1","1","1") )
```

`static Random()` → HP2.Matrix6 :

Return an object where all elements are randomly set to values between 0 and 1.

```
Zero = Matrix6( ("0","0","0","0","0","0"), ("0","0","0","0","0","0"),
("0","0","0","0","0","0"), ("0","0","0","0","0","0"),
("0","0","0","0","0","0"), ("0","0","0","0","0","0") )
```

__init__(*(object)arg1*) → None

___init___((object)arg1, (HP2.Matrix6)other) -> None

___init___((object)arg1, (HP2.Vector6)diag) -> object

___init___((object)arg1, (HP2.Matrix3)ul, (HP2.Matrix3)ur, (HP2.Matrix3)ll, (HP2.Matrix3)lr) -> object

___init___((object)arg1, (HP2.Vector6)l0, (HP2.Vector6)l1, (HP2.Vector6)l2, (HP2.Vector6)l3, (HP2.Vector6)l4, (HP2.Vector6)l5 [, (bool)cols=False]) -> object

col(*(HP2.Matrix6)arg1*, *(int)col*) → HP2.Vector6 :

Return column as vector.

cols(*(HP2.Matrix6)arg1*) → int :

Number of columns.

computeUnitaryPositive(*(HP2.Matrix6)arg1*) → tuple :

Compute polar decomposition (unitary matrix U and positive semi-definite symmetric matrix P such that self=U*P).

determinant(*(HP2.Matrix6)arg1*) → HP2.Real :

Return matrix determinant.

diagonal(*(HP2.Matrix6)arg1*) → HP2.Vector6 :

Return diagonal as vector.

inverse(*(HP2.Matrix6)arg1*) → HP2.Matrix6 :

Return inverted matrix.

isApprox(*(HP2.Matrix6)arg1*, *(HP2.Matrix6)other*[, *(HP2.Real)prec=Real("3.8427354393059617569828961866982853 64e-31")*]) → bool :

Approximate comparison with precision *prec*.

jacobiSVD(*(HP2.Matrix6)arg1*) → tuple :

Compute SVD decomposition of square matrix, retuns (U,S,V) such that self=U*S*V.transpose()

ll(*(HP2.Matrix6)arg1*) → HP2.Matrix3 :

Return lower-left 3x3 block

lr(*(HP2.Matrix6)arg1*) → HP2.Matrix3 :

Return lower-right 3x3 block

maxAbsCoeff(*(HP2.Matrix6)arg1*) → HP2.Real :

Maximum absolute value over all elements.

maxCoeff(*(HP2.Matrix6)arg1*) → HP2.Real :

Maximum value over all elements.

mean(*(HP2.Matrix6)arg1*) → HP2.Real :

Mean value over all elements.

minCoeff(*(HP2.Matrix6)arg1*) → HP2.Real :

Minimum value over all elements.

norm(*(HP2.Matrix6)arg1*) → HP2.Real :

Euclidean norm.

normalize(*(HP2.Matrix6)arg1*) → None :

Normalize this object in-place.

normalized(*(HP2.Matrix6)arg1*) → HP2.Matrix6 :

Return normalized copy of this object

polarDecomposition(*(HP2.Matrix6)arg1*) → tuple :

Alias for *computeUnitaryPositive*.

prod(*(HP2.Matrix6)arg1*) → HP2.Real :

Product of all elements.

pruned(*(HP2.Matrix6)arg1*[, *(float)absTol=1e-06*]) → HP2.Matrix6 :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

row(*(HP2.Matrix6)arg1*, *(int)row*) → HP2.Vector6 :

Return row as vector.

rows(*(HP2.Matrix6)arg1*) → int :

Number of rows.

selfAdjointEigenDecomposition(*(HP2.Matrix6)arg1*) → tuple :

Compute eigen (spectral) decomposition of symmetric matrix, returns (eigVecs,eigVals). eigVecs is orthogonal Matrix3 with columns ar normalized eigenvectors, eigVals is Vector3 with corresponding eigenvalues. self=eigVecs*diag(eigVals)*eigVecs.transpose().

spectralDecomposition(*(HP2.Matrix6)arg1*) → tuple :

Alias for *selfAdjointEigenDecomposition*.

squaredNorm(*(HP2.Matrix6)arg1*) → HP2.Real :

Square of the Euclidean norm.

sum(*(HP2.Matrix6)arg1*) → HP2.Real :

Sum of all elements.

svd(*(HP2.Matrix6)arg1*) → tuple :

Alias for *jacobiSVD*.

trace(*(HP2.Matrix6)arg1*) → HP2.Real :

Return sum of diagonal elements.

transpose(*(HP2.Matrix6)arg1*) → HP2.Matrix6 :

Return transposed matrix.

ul(*(HP2.Matrix6)arg1*) → HP2.Matrix3 :

Return upper-left 3x3 block

ur(*(HP2.Matrix6)arg1*) → HP2.Matrix3 :

Return upper-right 3x3 block

class Matrix6c

/TODO/

```
Identity = Matrix6c( (Complex("1","0"),Complex("0","0"),Complex("0","0"),
Complex("0","0"),Complex("0","0"),Complex("0","0")),
(Complex("0","0"),Complex("1","0"),Complex("0","0"),Complex("0","0"),
Complex("0","0"),Complex("0","0")),
(Complex("0","0"),Complex("0","0"),Complex("1","0"),Complex("0","0"),
Complex("0","0"),Complex("0","0")),
(Complex("0","0"),Complex("0","0"),Complex("0","0"),Complex("1","0"),
Complex("0","0"),Complex("0","0")),
(Complex("0","0"),Complex("0","0"),Complex("0","0"),Complex("0","0"),
Complex("1","0"),Complex("0","0")),
(Complex("0","0"),Complex("0","0"),Complex("0","0"),Complex("0","0"),
Complex("0","0"),Complex("1","0")) )
```

```
Ones = Matrix6c( (Complex("1","0"),Complex("1","0"),Complex("1","0"),
Complex("1","0"),Complex("1","0"),Complex("1","0")),
(Complex("1","0"),Complex("1","0"),Complex("1","0"),Complex("1","0"),
Complex("1","0"),Complex("1","0")),
(Complex("1","0"),Complex("1","0"),Complex("1","0"),Complex("1","0"),
Complex("1","0"),Complex("1","0")),
(Complex("1","0"),Complex("1","0"),Complex("1","0"),Complex("1","0"),
Complex("1","0"),Complex("1","0")),
(Complex("1","0"),Complex("1","0"),Complex("1","0"),Complex("1","0"),
Complex("1","0"),Complex("1","0")),
(Complex("1","0"),Complex("1","0"),Complex("1","0"),Complex("1","0"),
Complex("1","0"),Complex("1","0")) )
```

`static Random()` → HP2.Matrix6c :

Return an object where all elements are randomly set to values between 0 and 1.

```
Zero = Matrix6c( (Complex("0","0"),Complex("0","0"),Complex("0","0"),
Complex("0","0"),Complex("0","0"),Complex("0","0")),
(Complex("0","0"),Complex("0","0"),Complex("0","0"),Complex("0","0"),
Complex("0","0"),Complex("0","0")),
(Complex("0","0"),Complex("0","0"),Complex("0","0"),Complex("0","0"),
Complex("0","0"),Complex("0","0")),
(Complex("0","0"),Complex("0","0"),Complex("0","0"),Complex("0","0"),
Complex("0","0"),Complex("0","0")),
(Complex("0","0"),Complex("0","0"),Complex("0","0"),Complex("0","0"),
Complex("0","0"),Complex("0","0")),
(Complex("0","0"),Complex("0","0"),Complex("0","0"),Complex("0","0"),
Complex("0","0"),Complex("0","0")) )
```

`__init__`(*(object)arg1*) → None

___init___((object)arg1, (HP2.Matrix6c)other) -> None

___init___((object)arg1, (HP2.Vector6c)diag) -> object

___init___((object)arg1, (HP2.Matrix3c)ul, (HP2.Matrix3c)ur, (HP2.Matrix3c)ll, (HP2.Matrix3c)lr) -> object

___init___((object)arg1, (HP2.Vector6c)l0, (HP2.Vector6c)l1, (HP2.Vector6c)l2, (HP2.Vector6c)l3, (HP2.Vector6c)l4, (HP2.Vector6c)l5 [, (bool)cols=False]) -> object

`col`(*(HP2.Matrix6c)arg1*, *(int)col*) → HP2.Vector6c :

Return column as vector.

`cols`(*(HP2.Matrix6c)arg1*) → int :

Number of columns.

`determinant`(*(HP2.Matrix6c)arg1*) → HP2.Complex :

Return matrix determinant.

`diagonal`(*(HP2.Matrix6c)arg1*) → HP2.Vector6c :

Return diagonal as vector.

`inverse`(*(HP2.Matrix6c)arg1*) → HP2.Matrix6c :

Return inverted matrix.

`isApprox`(*(HP2.Matrix6c)arg1*, *(HP2.Matrix6c)other*[, *(HP2.Real)prec=Complex("3.84273543930596175698289618669828536 4e-31", "0")*]) → bool :

Approximate comparison with precision *prec*.

`ll`(*(HP2.Matrix6c)arg1*) → HP2.Matrix3c :
Return lower-left 3x3 block

`lr`(*(HP2.Matrix6c)arg1*) → HP2.Matrix3c :
Return lower-right 3x3 block

`maxAbsCoeff`(*(HP2.Matrix6c)arg1*) → HP2.Real :
Maximum absolute value over all elements.

`mean`(*(HP2.Matrix6c)arg1*) → HP2.Complex :
Mean value over all elements.

`norm`(*(HP2.Matrix6c)arg1*) → HP2.Real :
Euclidean norm.

`normalize`(*(HP2.Matrix6c)arg1*) → None :
Normalize this object in-place.

`normalized`(*(HP2.Matrix6c)arg1*) → HP2.Matrix6c :
Return normalized copy of this object

`prod`(*(HP2.Matrix6c)arg1*) → HP2.Complex :
Product of all elements.

`pruned`(*(HP2.Matrix6c)arg1*[, *(float)absTol=1e-06*]) → HP2.Matrix6c :
Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

`row`(*(HP2.Matrix6c)arg1*, *(int)row*) → HP2.Vector6c :
Return row as vector.

`rows`(*(HP2.Matrix6c)arg1*) → int :
Number of rows.

`squaredNorm`(*(HP2.Matrix6c)arg1*) → HP2.Real :
Square of the Euclidean norm.

`sum`(*(HP2.Matrix6c)arg1*) → HP2.Complex :
Sum of all elements.

`trace`(*(HP2.Matrix6c)arg1*) → HP2.Complex :
Return sum of diagonal elements.

`transpose`(*(HP2.Matrix6c)arg1*) → HP2.Matrix6c :
Return transposed matrix.

`ul`(*(HP2.Matrix6c)arg1*) → HP2.Matrix3c :
Return upper-left 3x3 block

`ur`(*(HP2.Matrix6c)arg1*) → HP2.Matrix3c :
Return upper-right 3x3 block

`class MatrixX`
XxX (dynamic-sized) float matrix. Constructed from list of rows (as VectorX).

Supported operations (`m` is a MatrixX, `f` if a float/int, `v` is a VectorX): `-m`, `m+m`, `m+=m`, `m-m`, `m-=m`, `m*f`, `f*m`, `m*=f`, `m/f`, `m/=f`, `m*m`, `m*=m`, `m*v`, `v*m`, `m==m`, `m!=m`.

`static Identity`(*(int)arg1*, *(int)rank*) → HP2.MatrixX :
Create identity matrix with given rank (square).

`static Ones`(*(int)rows*, *(int)cols*) → HP2.MatrixX :
Create matrix of given dimensions where all elements are set to 1.

static Random(*(int)rows*, *(int)cols*) → HP2.MatrixX :

Create matrix with given dimensions where all elements are set to number between 0 and 1 (uniformly-distributed).

static Zero(*(int)rows*, *(int)cols*) → HP2.MatrixX :

Create zero matrix of given dimensions

__init__(*(object)arg1*) → None

___init___((object)arg1, (HP2.MatrixX)other) -> None

___init___((object)arg1, (HP2.VectorX)diag) -> object

___init___((object)arg1 [, (HP2.VectorX)r0=VectorX() [, (HP2.VectorX)r1=VectorX() [, (HP2.VectorX)r2=VectorX() [, (HP2.VectorX)r3=VectorX() [, (HP2.VectorX)r4=VectorX() [, (HP2.VectorX)r5=VectorX() [, (HP2.VectorX)r6=VectorX() [, (HP2.VectorX)r7=VectorX() [, (HP2.VectorX)r8=VectorX() [, (HP2.VectorX)r9=VectorX() [, (bool)cols=False]]]]]]]]]]) -> object

___init___((object)arg1, (object)rows [, (bool)cols=False]) -> object

col(*(HP2.MatrixX)arg1*, *(int)col*) → HP2.VectorX :

Return column as vector.

cols(*(HP2.MatrixX)arg1*) → int :

Number of columns.

computeUnitaryPositive(*(HP2.MatrixX)arg1*) → tuple :

Compute polar decomposition (unitary matrix U and positive semi-definite symmetric matrix P such that self=U*P).

determinant(*(HP2.MatrixX)arg1*) → HP2.Real :

Return matrix determinant.

diagonal(*(HP2.MatrixX)arg1*) → HP2.VectorX :

Return diagonal as vector.

inverse(*(HP2.MatrixX)arg1*) → HP2.MatrixX :

Return inverted matrix.

isApprox(*(HP2.MatrixX)arg1*, *(HP2.MatrixX)other*[, *(HP2.Real)prec=Real("3.84273543930596175698289618669828536 4e-31")*]) → bool :

Approximate comparison with precision *prec*.

jacobiSVD(*(HP2.MatrixX)arg1*) → tuple :

Compute SVD decomposition of square matrix, retuns (U,S,V) such that self=U*S*V.transpose()

maxAbsCoeff(*(HP2.MatrixX)arg1*) → HP2.Real :

Maximum absolute value over all elements.

maxCoeff(*(HP2.MatrixX)arg1*) → HP2.Real :

Maximum value over all elements.

mean(*(HP2.MatrixX)arg1*) → HP2.Real :

Mean value over all elements.

minCoeff(*(HP2.MatrixX)arg1*) → HP2.Real :

Minimum value over all elements.

norm(*(HP2.MatrixX)arg1*) → HP2.Real :

Euclidean norm.

normalize(*(HP2.MatrixX)arg1*) → None :

Normalize this object in-place.

normalized(*(HP2.MatrixX)arg1*) → HP2.MatrixX :

Return normalized copy of this object

polarDecomposition(*(HP2.MatrixX)arg1*) → tuple :

Alias for *computeUnitaryPositive*.

prod(*(HP2.MatrixX)arg1*) → HP2.Real :

Product of all elements.

pruned(*(HP2.MatrixX)arg1*[, *(float)absTol=1e-06*]) → HP2.MatrixX :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

resize(*(HP2.MatrixX)arg1*, *(int)rows*, *(int)cols*) → None :

Change size of the matrix, keep values of elements which exist in the new matrix

row(*(HP2.MatrixX)arg1*, *(int)row*) → HP2.VectorX :

Return row as vector.

rows(*(HP2.MatrixX)arg1*) → int :

Number of rows.

selfAdjointEigenDecomposition(*(HP2.MatrixX)arg1*) → tuple :

Compute eigen (spectral) decomposition of symmetric matrix, returns (eigVecs,eigVals). eigVecs is orthogonal Matrix3 with columns ar normalized eigenvectors, eigVals is Vector3 with corresponding eigenvalues. self=eigVecs*diag(eigVals)*eigVecs.transpose().

spectralDecomposition(*(HP2.MatrixX)arg1*) → tuple :

Alias for *selfAdjointEigenDecomposition*.

squaredNorm(*(HP2.MatrixX)arg1*) → HP2.Real :

Square of the Euclidean norm.

sum(*(HP2.MatrixX)arg1*) → HP2.Real :

Sum of all elements.

svd(*(HP2.MatrixX)arg1*) → tuple :

Alias for *jacobiSVD*.

trace(*(HP2.MatrixX)arg1*) → HP2.Real :

Return sum of diagonal elements.

transpose(*(HP2.MatrixX)arg1*) → HP2.MatrixX :

Return transposed matrix.

class MatrixXc

/TODO/

static Identity(*(int)arg1*, *(int)rank*) → HP2.MatrixXc :

Create identity matrix with given rank (square).

static Ones(*(int)rows*, *(int)cols*) → HP2.MatrixXc :

Create matrix of given dimensions where all elements are set to 1.

static Random(*(int)rows*, *(int)cols*) → HP2.MatrixXc :

Create matrix with given dimensions where all elements are set to number between 0 and 1 (uniformly-distributed).

static Zero(*(int)rows*, *(int)cols*) → HP2.MatrixXc :

Create zero matrix of given dimensions

__init__(*(object)arg1*) → None

___init___((object)arg1, (HP2.MatrixXc)other) -> None

___init___((object)arg1, (HP2.VectorXc)diag) -> object

___init___((object)arg1 [, (HP2.VectorXc)r0=VectorXc() [, (HP2.VectorXc)r1=VectorXc() [, (HP2.VectorXc)r2=VectorXc() [, (HP2.VectorXc)r3=VectorXc() [, (HP2.VectorXc)r4=VectorXc() [, (HP2.VectorXc)r5=VectorXc() [, (HP2.VectorXc)r6=VectorXc() [, (HP2.VectorXc)r7=VectorXc() [, (HP2.VectorXc)r8=VectorXc() [, (HP2.VectorXc)r9=VectorXc() [, (bool)cols=False]]]]]]]]]]) -> object

___init___((object)arg1, (object)rows [, (bool)cols=False]) -> object

col(*(HP2.MatrixXc)arg1*, *(int)col*) → HP2.VectorXc :

Return column as vector.

cols(*(HP2.MatrixXc)arg1*) → int :

Number of columns.

determinant(*(HP2.MatrixXc)arg1*) → HP2.Complex :

Return matrix determinant.

diagonal(*(HP2.MatrixXc)arg1*) → HP2.VectorXc :

Return diagonal as vector.

inverse(*(HP2.MatrixXc)arg1*) → HP2.MatrixXc :

Return inverted matrix.

isApprox(*(HP2.MatrixXc)arg1*, *(HP2.MatrixXc)other*[, *(HP2.Real)prec=Complex("3.842735439305961756982896186698285364e-31", "0")*]) → bool :

Approximate comparison with precision *prec*.

maxAbsCoeff(*(HP2.MatrixXc)arg1*) → HP2.Real :

Maximum absolute value over all elements.

mean(*(HP2.MatrixXc)arg1*) → HP2.Complex :

Mean value over all elements.

norm(*(HP2.MatrixXc)arg1*) → HP2.Real :

Euclidean norm.

normalize(*(HP2.MatrixXc)arg1*) → None :

Normalize this object in-place.

normalized(*(HP2.MatrixXc)arg1*) → HP2.MatrixXc :

Return normalized copy of this object

prod(*(HP2.MatrixXc)arg1*) → HP2.Complex :

Product of all elements.

pruned(*(HP2.MatrixXc)arg1*[, *(float)absTol=1e-06*]) → HP2.MatrixXc :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

resize(*(HP2.MatrixXc)arg1*, *(int)rows*, *(int)cols*) → None :

Change size of the matrix, keep values of elements which exist in the new matrix

row(*(HP2.MatrixXc)arg1*, *(int)row*) → HP2.VectorXc :

Return row as vector.

rows(*(HP2.MatrixXc)arg1*) → int :

Number of rows.

`squaredNorm`(*(HP2.MatrixXc)arg1*) → HP2.Real :

Square of the Euclidean norm.

`sum`(*(HP2.MatrixXc)arg1*) → HP2.Complex :

Sum of all elements.

`trace`(*(HP2.MatrixXc)arg1*) → HP2.Complex :

Return sum of diagonal elements.

`transpose`(*(HP2.MatrixXc)arg1*) → HP2.MatrixXc :

Return transposed matrix.

`class Quaternion`

Quaternion representing rotation.

Supported operations (`q` is a Quaternion, `v` is a Vector3): `q*q` (rotation composition), `q*=q`, `q*v` (rotating `v` by `q`), `q==q`, `q!=q`.

Static attributes: `Identity`.

Note

Quaternion is represented as axis-angle when printed (e.g. `Identity` is `Quaternion((1,0,0),0)`, and can also be constructed from the axis-angle representation. This is however different from the data stored inside, which can be accessed by indices `[0]` (`x`), `[1]` (`y`), `[2]` (`z`), `[3]` (*w*). To obtain axis-angle programatically, use *`Quaternion.toAxisAngle`* which returns the tuple.

`Identity = Quaternion(("1","0","0"),"0")`

`Rotate`(*(HP2.Quaternion)arg1*, *(HP2.Vector3)v*) → *HP2.Vector3*

`__init__`(*(object)arg1*) → None

___init___((object)arg1, (HP2.Vector3)axis, (object)angle) -> object

___init___((object)arg1, (HP2.Vector3)axis, (HP2.Real)angle) -> object

___init___((object)arg1, (object)angle, (HP2.Vector3)axis) -> object

___init___((object)arg1, (HP2.Real)angle, (HP2.Vector3)axis) -> object

___init___((object)arg1, (tuple)axis, (str)angle) -> object

___init___((object)arg1, (tuple)tuple) -> object

___init___((object)arg1, (HP2.Vector3)u, (HP2.Vector3)v) -> object

___init___((object)arg1, (str)str1, (str)str2, (str)str3, (str)str4) -> object

___init___((object)arg1, (HP2.Real)w, (HP2.Real)x, (HP2.Real)y, (HP2.Real)z) -> None :

Initialize from coefficients.

Note

The order of coefficients is *w*, *x*, *y*, *z*. The [] operator numbers them differently, 0...4 for *x y z w*!

___init___((object)arg1, (HP2.Matrix3)rotMatrix) -> None

___init___((object)arg1, (HP2.Quaternion)other) -> None

___init___((object)arg1, (float)w, (float)x, (float)y, (float)z) -> None :

Initialize from coefficients.

Note

The order of coefficients is w, x, y, z. The [] operator numbers them differently, 0...4 for x y z w!

angularDistance(*(HP2.Quaternion)arg1*, *(HP2.Quaternion)arg2*) → *HP2.Real*

conjugate(*(HP2.Quaternion)arg1*) → *HP2.Quaternion*

inverse(*(HP2.Quaternion)arg1*) → *HP2.Quaternion*

norm(*(HP2.Quaternion)arg1*) → *HP2.Real*

normalize(*(HP2.Quaternion)arg1*) → None

normalized(*(HP2.Quaternion)arg1*) → *HP2.Quaternion*

setFromTwoVectors(*(HP2.Quaternion)arg1*, *(HP2.Vector3)u*, *(HP2.Vector3)v*) → None

slerp(*(HP2.Quaternion)arg1*, *(HP2.Real)t*, *(HP2.Quaternion)other*) → *HP2.Quaternion*

toAngleAxis(*(HP2.Quaternion)arg1*) → tuple

toAxisAngle(*(HP2.Quaternion)arg1*) → tuple

toRotationMatrix(*(HP2.Quaternion)arg1*) → *HP2.Matrix3*

toRotationVector(*(HP2.Quaternion)arg1*) → *HP2.Vector3*

class Real

The Real type.

__init__(*(object)arg1*) → None

___init___((object)arg1, (object)obj) -> object

___init___((object)arg1, (float)d) -> object

___init___((object)arg1, (int)i) -> object

___init___((object)arg1, (str)str) -> object

property imag

None((yade._minieigenHP.HP2.Real)arg1) -> yade._minieigenHP.HP2.Real

property levelHP

None((yade._minieigenHP.HP2.Real)arg1) -> int

levelRealHPMethod(*(HP2.Real)arg1*) → int

property real

None((yade._minieigenHP.HP2.Real)arg1) -> yade._minieigenHP.HP2.Real

sqrt(*(HP2.Real)arg1*) → *HP2.Real*

class Vector2

3-dimensional float vector.

Supported operations (`f` if a float/int, `v` is a Vector3): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`.

Implicit conversion from sequence (list, tuple, ...) of 2 floats.

Static attributes: `Zero`, `Ones`, `UnitX`, `UnitY`.

```
Identity = Vector2("1","0")
```

Ones = Vector2("1","1")

static Random() → HP2.Vector2 :

Return an object where all elements are randomly set to values between 0 and 1.

static Unit(*(int)arg1***)** → *HP2.Vector2*

UnitX = Vector2("1","0")

UnitY = Vector2("0","1")

Zero = Vector2("0","0")

__init__(*(object)arg1***)** → None

___init___((object)arg1, (HP2.Vector2)other) -> None

___init___((object)arg1, (str)str1, (str)str2) -> object

___init___((object)arg1, (HP2.Real)x, (HP2.Real)y) -> None

asDiagonal(*(HP2.Vector2)arg1***)** → object :

Return diagonal matrix with this vector on the diagonal.

cols(*(HP2.Vector2)arg1***)** → int :

Number of columns.

dot(*(HP2.Vector2)arg1*, *(HP2.Vector2)other***)** → HP2.Real :

Dot product with *other*.

isApprox(*(HP2.Vector2)arg1*, *(HP2.Vector2)other*[, *(HP2.Real)prec=Real("3.84273543930596175698289618669828536 4e-31")*]**)** → bool :

Approximate comparison with precision *prec*.

maxAbsCoeff(*(HP2.Vector2)arg1***)** → HP2.Real :

Maximum absolute value over all elements.

maxCoeff(*(HP2.Vector2)arg1***)** → HP2.Real :

Maximum value over all elements.

mean(*(HP2.Vector2)arg1***)** → HP2.Real :

Mean value over all elements.

minCoeff(*(HP2.Vector2)arg1***)** → HP2.Real :

Minimum value over all elements.

norm(*(HP2.Vector2)arg1***)** → HP2.Real :

Euclidean norm.

normalize(*(HP2.Vector2)arg1***)** → None :

Normalize this object in-place.

normalized(*(HP2.Vector2)arg1***)** → HP2.Vector2 :

Return normalized copy of this object

outer(*(HP2.Vector2)arg1*, *(HP2.Vector2)other***)** → object :

Outer product with *other*.

prod(*(HP2.Vector2)arg1***)** → HP2.Real :

Product of all elements.

pruned(*(HP2.Vector2)arg1*[, *(float)absTol=1e-06*]**)** → HP2.Vector2 :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

rows(*(HP2.Vector2)arg1*) → int :

Number of rows.

squaredNorm(*(HP2.Vector2)arg1*) → HP2.Real :

Square of the Euclidean norm.

sum(*(HP2.Vector2)arg1*) → HP2.Real :

Sum of all elements.

class Vector2c

/TODO/

Identity = Vector2c(Complex("1","0"),Complex("0","0"))

Ones = Vector2c(Complex("1","0"),Complex("1","0"))

static Random() → HP2.Vector2c :

Return an object where all elements are randomly set to values between 0 and 1.

static Unit(*(int)arg1*) → *HP2.Vector2c*

UnitX = Vector2c(Complex("1","0"),Complex("0","0"))

UnitY = Vector2c(Complex("0","0"),Complex("1","0"))

Zero = Vector2c(Complex("0","0"),Complex("0","0"))

__init__(*(object)arg1*) → None

___init___((object)arg1, (HP2.Vector2c)other) -> None

___init___((object)arg1, (str)str1, (str)str2) -> object

___init___((object)arg1, (HP2.Complex)x, (HP2.Complex)y) -> None

asDiagonal(*(HP2.Vector2c)arg1*) → object :

Return diagonal matrix with this vector on the diagonal.

cols(*(HP2.Vector2c)arg1*) → int :

Number of columns.

dot(*(HP2.Vector2c)arg1*, *(HP2.Vector2c)other*) → HP2.Complex :

Dot product with *other*.

isApprox(*(HP2.Vector2c)arg1*, *(HP2.Vector2c)other*[, *(HP2.Real)prec=Complex("3.842735439305961756982896186698285364e-31", "0")*]) → bool :

Approximate comparison with precision *prec*.

maxAbsCoeff(*(HP2.Vector2c)arg1*) → HP2.Real :

Maximum absolute value over all elements.

mean(*(HP2.Vector2c)arg1*) → HP2.Complex :

Mean value over all elements.

norm(*(HP2.Vector2c)arg1*) → HP2.Real :

Euclidean norm.

normalize(*(HP2.Vector2c)arg1*) → None :

Normalize this object in-place.

normalized(*(HP2.Vector2c)arg1*) → HP2.Vector2c :

Return normalized copy of this object

`outer`(*(HP2.Vector2c)arg1*, *(HP2.Vector2c)other*) → object :

Outer product with *other*.

`prod`(*(HP2.Vector2c)arg1*) → HP2.Complex :

Product of all elements.

`pruned`(*(HP2.Vector2c)arg1*[, *(float)absTol=1e-06*]) → HP2.Vector2c :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

`rows`(*(HP2.Vector2c)arg1*) → int :

Number of rows.

`squaredNorm`(*(HP2.Vector2c)arg1*) → HP2.Real :

Square of the Euclidean norm.

`sum`(*(HP2.Vector2c)arg1*) → HP2.Complex :

Sum of all elements.

`class Vector2i`

2-dimensional integer vector.

Supported operations (`i` if an int, `v` is a Vector2i): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*i`, `i*v`, `v*=i`, `v==v`, `v!=v`.

Implicit conversion from sequence (list, tuple, ...) of 2 integers.

Static attributes: `Zero`, `Ones`, `UnitX`, `UnitY`.

`Identity = Vector2i(1,0)`

`Ones = Vector2i(1,1)`

`static Random()` → Vector2i :

Return an object where all elements are randomly set to values between 0 and 1.

`static Unit`(*(int)arg1*) → *Vector2i*

`UnitX = Vector2i(1,0)`

`UnitY = Vector2i(0,1)`

`Zero = Vector2i(0,0)`

`__init__`(*(object)arg1*) → None

___init___((object)arg1, (Vector2i)other) -> None

___init___((object)arg1, (str)str1, (str)str2) -> object

___init___((object)arg1, (int)x, (int)y) -> None

`asDiagonal`(*(Vector2i)arg1*) → object :

Return diagonal matrix with this vector on the diagonal.

`cols`(*(Vector2i)arg1*) → int :

Number of columns.

`dot`(*(Vector2i)arg1*, *(Vector2i)other*) → int :

Dot product with *other*.

`isApprox`(*(Vector2i)arg1*, *(Vector2i)other*[, *(int)prec=0*]) → bool :

Approximate comparison with precision *prec*.

`maxAbsCoeff`(*(Vector2i)arg1*) → int :

Maximum absolute value over all elements.

maxCoeff(*(Vector2i)arg1*) → int :

Maximum value over all elements.

mean(*(Vector2i)arg1*) → int :

Mean value over all elements.

minCoeff(*(Vector2i)arg1*) → int :

Minimum value over all elements.

outer(*(Vector2i)arg1*, *(Vector2i)other*) → object :

Outer product with *other*.

prod(*(Vector2i)arg1*) → int :

Product of all elements.

rows(*(Vector2i)arg1*) → int :

Number of rows.

sum(*(Vector2i)arg1*) → int :

Sum of all elements.

class Vector3

3-dimensional float vector.

Supported operations (`f` if a float/int, `v` is a Vector3): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`, plus operations with `Matrix3` and `Quaternion`.

Implicit conversion from sequence (list, tuple, ...) of 3 floats.

Static attributes: `Zero`, `Ones`, `UnitX`, `UnitY`, `UnitZ`.

Identity = Vector3("1","0","0")

Ones = Vector3("1","1","1")

static Random() → HP2.Vector3 :

Return an object where all elements are randomly set to values between 0 and 1.

static Unit(*(int)arg1*) → *HP2.Vector3*

UnitX = Vector3("1","0","0")

UnitY = Vector3("0","1","0")

UnitZ = Vector3("0","0","1")

Zero = Vector3("0","0","0")

__init__(*(object)arg1*) → None

___init___((object)arg1, (HP2.Vector3)other) -> None

___init___((object)arg1, (str)str1, (str)str2, (str)str3) -> object

___init___((object)arg1 [, (HP2.Real)x=Real("0") [, (HP2.Real)y=Real("0") [, (HP2.Real)z=Real("0")]]]) -> None

asDiagonal(*(HP2.Vector3)arg1*) → HP2.Matrix3 :

Return diagonal matrix with this vector on the diagonal.

cols(*(HP2.Vector3)arg1*) → int :

Number of columns.

cross(*(HP2.Vector3)arg1*, *(HP2.Vector3)arg2*) → *HP2.Vector3*

`dot`(*(HP2.Vector3)arg1*, *(HP2.Vector3)other*) → HP2.Real :
Dot product with *other*.

`isApprox`(*(HP2.Vector3)arg1*, *(HP2.Vector3)other*[, *(HP2.Real)prec=Real("3.84273543930596175698289618669828536 4e-31")*]) → bool :
Approximate comparison with precision *prec*.

`maxAbsCoeff`(*(HP2.Vector3)arg1*) → HP2.Real :
Maximum absolute value over all elements.

`maxCoeff`(*(HP2.Vector3)arg1*) → HP2.Real :
Maximum value over all elements.

`mean`(*(HP2.Vector3)arg1*) → HP2.Real :
Mean value over all elements.

`minCoeff`(*(HP2.Vector3)arg1*) → HP2.Real :
Minimum value over all elements.

`norm`(*(HP2.Vector3)arg1*) → HP2.Real :
Euclidean norm.

`normalize`(*(HP2.Vector3)arg1*) → None :
Normalize this object in-place.

`normalized`(*(HP2.Vector3)arg1*) → HP2.Vector3 :
Return normalized copy of this object

`outer`(*(HP2.Vector3)arg1*, *(HP2.Vector3)other*) → HP2.Matrix3 :
Outer product with *other*.

`prod`(*(HP2.Vector3)arg1*) → HP2.Real :
Product of all elements.

`pruned`(*(HP2.Vector3)arg1*[, *(float)absTol=1e-06*]) → HP2.Vector3 :
Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

`rows`(*(HP2.Vector3)arg1*) → int :
Number of rows.

`squaredNorm`(*(HP2.Vector3)arg1*) → HP2.Real :
Square of the Euclidean norm.

`sum`(*(HP2.Vector3)arg1*) → HP2.Real :
Sum of all elements.

`xy`(*(HP2.Vector3)arg1*) → *HP2.Vector2*

`xz`(*(HP2.Vector3)arg1*) → *HP2.Vector2*

`yx`(*(HP2.Vector3)arg1*) → *HP2.Vector2*

`yz`(*(HP2.Vector3)arg1*) → *HP2.Vector2*

`zx`(*(HP2.Vector3)arg1*) → *HP2.Vector2*

`zy`(*(HP2.Vector3)arg1*) → *HP2.Vector2*

`class Vector3c`
/TODO/

Identity = Vector3c(Complex("1","0"),Complex("0","0"),Complex("0","0"))

Ones = Vector3c(Complex("1","0"),Complex("1","0"),Complex("1","0"))

static Random() → HP2.Vector3c :

Return an object where all elements are randomly set to values between 0 and 1.

static Unit(*(int)arg1*) → *HP2.Vector3c*

UnitX = Vector3c(Complex("1","0"),Complex("0","0"),Complex("0","0"))

UnitY = Vector3c(Complex("0","0"),Complex("1","0"),Complex("0","0"))

UnitZ = Vector3c(Complex("0","0"),Complex("0","0"),Complex("1","0"))

Zero = Vector3c(Complex("0","0"),Complex("0","0"),Complex("0","0"))

__init__(*(object)arg1*) → None

___init___((object)arg1, (HP2.Vector3c)other) -> None

___init___((object)arg1, (str)str1, (str)str2, (str)str3) -> object

___init___((object)arg1 [, (HP2.Complex)x=Complex("0","0") [, (HP2.Complex)y=Complex("0","0") [, (HP2.Complex)z=Complex("0","0")]]]) -> None

asDiagonal(*(HP2.Vector3c)arg1*) → HP2.Matrix3c :

Return diagonal matrix with this vector on the diagonal.

cols(*(HP2.Vector3c)arg1*) → int :

Number of columns.

cross(*(HP2.Vector3c)arg1*, *(HP2.Vector3c)arg2*) → *HP2.Vector3c*

dot(*(HP2.Vector3c)arg1*, *(HP2.Vector3c)other*) → HP2.Complex :

Dot product with *other*.

isApprox(*(HP2.Vector3c)arg1*, *(HP2.Vector3c)other*[, *(HP2.Real)prec=Complex("3.842735439305961756982896186698285364e-31", "0")*]) → bool :

Approximate comparison with precision *prec*.

maxAbsCoeff(*(HP2.Vector3c)arg1*) → HP2.Real :

Maximum absolute value over all elements.

mean(*(HP2.Vector3c)arg1*) → HP2.Complex :

Mean value over all elements.

norm(*(HP2.Vector3c)arg1*) → HP2.Real :

Euclidean norm.

normalize(*(HP2.Vector3c)arg1*) → None :

Normalize this object in-place.

normalized(*(HP2.Vector3c)arg1*) → HP2.Vector3c :

Return normalized copy of this object

outer(*(HP2.Vector3c)arg1*, *(HP2.Vector3c)other*) → HP2.Matrix3c :

Outer product with *other*.

prod(*(HP2.Vector3c)arg1*) → HP2.Complex :

Product of all elements.

pruned(*(HP2.Vector3c)arg1*[, *(float)absTol=1e-06*]) → HP2.Vector3c :
Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

rows(*(HP2.Vector3c)arg1*) → int :
Number of rows.

squaredNorm(*(HP2.Vector3c)arg1*) → HP2.Real :
Square of the Euclidean norm.

sum(*(HP2.Vector3c)arg1*) → HP2.Complex :
Sum of all elements.

xy(*(HP2.Vector3c)arg1*) → *HP2.Vector2c*

xz(*(HP2.Vector3c)arg1*) → *HP2.Vector2c*

yx(*(HP2.Vector3c)arg1*) → *HP2.Vector2c*

yz(*(HP2.Vector3c)arg1*) → *HP2.Vector2c*

zx(*(HP2.Vector3c)arg1*) → *HP2.Vector2c*

zy(*(HP2.Vector3c)arg1*) → *HP2.Vector2c*

class Vector3i
3-dimensional integer vector.

Supported operations (`i` if an int, `v` is a Vector3i): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*i`, `i*v`, `v*=i`, `v==v`, `v!=v`.

Implicit conversion from sequence (list, tuple, ...) of 3 integers.

Static attributes: **Zero**, **Ones**, **UnitX**, **UnitY**, **UnitZ**.

Identity = Vector3i(1,0,0)

Ones = Vector3i(1,1,1)

static Random() → Vector3i :
Return an object where all elements are randomly set to values between 0 and 1.

static Unit(*(int)arg1*) → *Vector3i*

UnitX = Vector3i(1,0,0)

UnitY = Vector3i(0,1,0)

UnitZ = Vector3i(0,0,1)

Zero = Vector3i(0,0,0)

__init__(*(object)arg1*) → None
___init___((object)arg1, (Vector3i)other) -> None

___init___((object)arg1, (str)str1, (str)str2, (str)str3) -> object

___init___((object)arg1 [, (int)x=0 [, (int)y=0 [, (int)z=0]]]) -> None

asDiagonal(*(Vector3i)arg1*) → object :
Return diagonal matrix with this vector on the diagonal.

cols(*(Vector3i)arg1*) → int :
Number of columns.

cross(*(Vector3i)arg1*, *(Vector3i)arg2*) → *Vector3i*

`dot`(*(Vector3i)arg1*, *(Vector3i)other*) → int :
Dot product with *other*.

`isApprox`(*(Vector3i)arg1*, *(Vector3i)other*[, *(int)prec=0*]) → bool :
Approximate comparison with precision *prec*.

`maxAbsCoeff`(*(Vector3i)arg1*) → int :
Maximum absolute value over all elements.

`maxCoeff`(*(Vector3i)arg1*) → int :
Maximum value over all elements.

`mean`(*(Vector3i)arg1*) → int :
Mean value over all elements.

`minCoeff`(*(Vector3i)arg1*) → int :
Minimum value over all elements.

`outer`(*(Vector3i)arg1*, *(Vector3i)other*) → object :
Outer product with *other*.

`prod`(*(Vector3i)arg1*) → int :
Product of all elements.

`rows`(*(Vector3i)arg1*) → int :
Number of rows.

`sum`(*(Vector3i)arg1*) → int :
Sum of all elements.

`xy`(*(Vector3i)arg1*) → *Vector2i*

`xz`(*(Vector3i)arg1*) → *Vector2i*

`yx`(*(Vector3i)arg1*) → *Vector2i*

`yz`(*(Vector3i)arg1*) → *Vector2i*

`zx`(*(Vector3i)arg1*) → *Vector2i*

`zy`(*(Vector3i)arg1*) → *Vector2i*

`class Vector4`
4-dimensional float vector.

Supported operations (`f` if a float/int, `v` is a Vector3): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`.

Implicit conversion from sequence (list, tuple, ...) of 4 floats.

Static attributes: `Zero`, `Ones`.

`Identity = Vector4("1","0","0", "0")`

`Ones = Vector4("1","1","1", "1")`

`static Random()` → HP2.Vector4 :
Return an object where all elements are randomly set to values between 0 and 1.

`static Unit`(*(int)arg1*) → *HP2.Vector4*

`Zero = Vector4("0","0","0", "0")`

__init__(*(object)arg1*) → None

___init___((object)arg1, (HP2.Vector4)other) -> None

___init___((object)arg1, (str)str1, (str)str2, (str)str3, (str)str4) -> object

___init___((object)arg1, (HP2.Real)v0, (HP2.Real)v1, (HP2.Real)v2, (HP2.Real)v3) -> None

asDiagonal(*(HP2.Vector4)arg1*) → object :

Return diagonal matrix with this vector on the diagonal.

cols(*(HP2.Vector4)arg1*) → int :

Number of columns.

dot(*(HP2.Vector4)arg1*, *(HP2.Vector4)other*) → HP2.Real :

Dot product with *other*.

isApprox(*(HP2.Vector4)arg1*, *(HP2.Vector4)other*[, *(HP2.Real)prec=Real("3.84273543930596175698289618669828536 4e-31")*]) → bool :

Approximate comparison with precision *prec*.

maxAbsCoeff(*(HP2.Vector4)arg1*) → HP2.Real :

Maximum absolute value over all elements.

maxCoeff(*(HP2.Vector4)arg1*) → HP2.Real :

Maximum value over all elements.

mean(*(HP2.Vector4)arg1*) → HP2.Real :

Mean value over all elements.

minCoeff(*(HP2.Vector4)arg1*) → HP2.Real :

Minimum value over all elements.

norm(*(HP2.Vector4)arg1*) → HP2.Real :

Euclidean norm.

normalize(*(HP2.Vector4)arg1*) → None :

Normalize this object in-place.

normalized(*(HP2.Vector4)arg1*) → HP2.Vector4 :

Return normalized copy of this object

outer(*(HP2.Vector4)arg1*, *(HP2.Vector4)other*) → object :

Outer product with *other*.

prod(*(HP2.Vector4)arg1*) → HP2.Real :

Product of all elements.

pruned(*(HP2.Vector4)arg1*[, *(float)absTol=1e-06*]) → HP2.Vector4 :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

rows(*(HP2.Vector4)arg1*) → int :

Number of rows.

squaredNorm(*(HP2.Vector4)arg1*) → HP2.Real :

Square of the Euclidean norm.

sum(*(HP2.Vector4)arg1*) → HP2.Real :

Sum of all elements.

`class Vector6`

6-dimensional float vector.

Supported operations (`f` if a float/int, `v` is a Vector6): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`.

Implicit conversion from sequence (list, tuple, ...) of 6 floats.

Static attributes: `Zero`, `Ones`.

`Identity = Vector6("1","0","0", "0","0","0")`

`Ones = Vector6("1","1","1", "1","1","1")`

`static Random()` → HP2.Vector6 :

Return an object where all elements are randomly set to values between 0 and 1.

`static Unit(`*(int)arg1*`)` → *HP2.Vector6*

`Zero = Vector6("0","0","0", "0","0","0")`

`__init__(`*(object)arg1*`)` → None

___init___((object)arg1, (HP2.Vector6)other) -> None

___init___((object)arg1, (HP2.Real)v0, (HP2.Real)v1, (HP2.Real)v2, (HP2.Real)v3, (HP2.Real)v4, (HP2.Real)v5) -> object

___init___((object)arg1, (HP2.Vector3)head, (HP2.Vector3)tail) -> object

`asDiagonal(`*(HP2.Vector6)arg1*`)` → HP2.Matrix6 :

Return diagonal matrix with this vector on the diagonal.

`cols(`*(HP2.Vector6)arg1*`)` → int :

Number of columns.

`dot(`*(HP2.Vector6)arg1*, *(HP2.Vector6)other*`)` → HP2.Real :

Dot product with *other*.

`head(`*(HP2.Vector6)arg1*`)` → *HP2.Vector3*

`isApprox(`*(HP2.Vector6)arg1*, *(HP2.Vector6)other*[, *(HP2.Real)prec=Real("3.8427354393059617569828961866982853 64e-31")*]`)` → bool :

Approximate comparison with precision *prec*.

`maxAbsCoeff(`*(HP2.Vector6)arg1*`)` → HP2.Real :

Maximum absolute value over all elements.

`maxCoeff(`*(HP2.Vector6)arg1*`)` → HP2.Real :

Maximum value over all elements.

`mean(`*(HP2.Vector6)arg1*`)` → HP2.Real :

Mean value over all elements.

`minCoeff(`*(HP2.Vector6)arg1*`)` → HP2.Real :

Minimum value over all elements.

`norm(`*(HP2.Vector6)arg1*`)` → HP2.Real :

Euclidean norm.

`normalize(`*(HP2.Vector6)arg1*`)` → None :

Normalize this object in-place.

`normalized`(*(HP2.Vector6)arg1*) → HP2.Vector6 :

Return normalized copy of this object

`outer`(*(HP2.Vector6)arg1*, *(HP2.Vector6)other*) → HP2.Matrix6 :

Outer product with *other*.

`prod`(*(HP2.Vector6)arg1*) → HP2.Real :

Product of all elements.

`pruned`(*(HP2.Vector6)arg1*[, *(float)absTol=1e-06*]) → HP2.Vector6 :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

`rows`(*(HP2.Vector6)arg1*) → int :

Number of rows.

`squaredNorm`(*(HP2.Vector6)arg1*) → HP2.Real :

Square of the Euclidean norm.

`sum`(*(HP2.Vector6)arg1*) → HP2.Real :

Sum of all elements.

`tail`(*(HP2.Vector6)arg1*) → *HP2.Vector3*

`class Vector6c`

/TODO/

```
Identity = Vector6c(Complex("1","0"),Complex("0","0"),Complex("0","0"),
Complex("0","0"),Complex("0","0"),Complex("0","0"))
```

```
Ones = Vector6c(Complex("1","0"),Complex("1","0"),Complex("1","0"),
Complex("1","0"),Complex("1","0"),Complex("1","0"))
```

`static Random`() → HP2.Vector6c :

Return an object where all elements are randomly set to values between 0 and 1.

`static Unit`(*(int)arg1*) → *HP2.Vector6c*

```
Zero = Vector6c(Complex("0","0"),Complex("0","0"),Complex("0","0"),
Complex("0","0"),Complex("0","0"),Complex("0","0"))
```

`__init__`(*(object)arg1*) → None

___init___((object)arg1, (HP2.Vector6c)other) -> None

___init___((object)arg1, (HP2.Complex)v0, (HP2.Complex)v1, (HP2.Complex)v2, (HP2.Complex)v3, (HP2.Complex)v4, (HP2.Complex)v5) -> object

___init___((object)arg1, (HP2.Vector3c)head, (HP2.Vector3c)tail) -> object

`asDiagonal`(*(HP2.Vector6c)arg1*) → HP2.Matrix6c :

Return diagonal matrix with this vector on the diagonal.

`cols`(*(HP2.Vector6c)arg1*) → int :

Number of columns.

`dot`(*(HP2.Vector6c)arg1*, *(HP2.Vector6c)other*) → HP2.Complex :

Dot product with *other*.

`head`(*(HP2.Vector6c)arg1*) → *HP2.Vector3c*

`isApprox`(*(HP2.Vector6c)arg1*, *(HP2.Vector6c)other*[, *(HP2.Real)prec=Complex("3.842735439305961756982896186698285364e-31", "0")*]) → bool :

Approximate comparison with precision *prec*.

maxAbsCoeff(*(HP2.Vector6c)arg1*) → HP2.Real :
Maximum absolute value over all elements.

mean(*(HP2.Vector6c)arg1*) → HP2.Complex :
Mean value over all elements.

norm(*(HP2.Vector6c)arg1*) → HP2.Real :
Euclidean norm.

normalize(*(HP2.Vector6c)arg1*) → None :
Normalize this object in-place.

normalized(*(HP2.Vector6c)arg1*) → HP2.Vector6c :
Return normalized copy of this object

outer(*(HP2.Vector6c)arg1*, *(HP2.Vector6c)other*) → HP2.Matrix6c :
Outer product with *other*.

prod(*(HP2.Vector6c)arg1*) → HP2.Complex :
Product of all elements.

pruned(*(HP2.Vector6c)arg1*[, *(float)absTol=1e-06*]) → HP2.Vector6c :
Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

rows(*(HP2.Vector6c)arg1*) → int :
Number of rows.

squaredNorm(*(HP2.Vector6c)arg1*) → HP2.Real :
Square of the Euclidean norm.

sum(*(HP2.Vector6c)arg1*) → HP2.Complex :
Sum of all elements.

tail(*(HP2.Vector6c)arg1*) → *HP2.Vector3c*

class Vector6i
6-dimensional float vector.

Supported operations (`f` if a float/int, `v` is a Vector6): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`.

Implicit conversion from sequence (list, tuple, …) of 6 ints.

Static attributes: `Zero`, `Ones`.

Identity = Vector6i(1,0,0, 0,0,0)

Ones = Vector6i(1,1,1, 1,1,1)

static Random() → Vector6i :
Return an object where all elements are randomly set to values between 0 and 1.

static Unit(*(int)arg1*) → *Vector6i*

Zero = Vector6i(0,0,0, 0,0,0)

__init__(*(object)arg1*) → None
___init___((object)arg1, (Vector6i)other) -> None

___init___((object)arg1, (int)v0, (int)v1, (int)v2, (int)v3, (int)v4, (int)v5) -> object

___init___((object)arg1, (Vector3i)head, (Vector3i)tail) -> object

asDiagonal(*(Vector6i)arg1*) → object :
Return diagonal matrix with this vector on the diagonal.

`cols`(*(Vector6i)arg1*) → int :
Number of columns.

`dot`(*(Vector6i)arg1*, *(Vector6i)other*) → int :
Dot product with *other*.

`head`(*(Vector6i)arg1*) → *Vector3i*

`isApprox`(*(Vector6i)arg1*, *(Vector6i)other*[, *(int)prec=0*]) → bool :
Approximate comparison with precision *prec*.

`maxAbsCoeff`(*(Vector6i)arg1*) → int :
Maximum absolute value over all elements.

`maxCoeff`(*(Vector6i)arg1*) → int :
Maximum value over all elements.

`mean`(*(Vector6i)arg1*) → int :
Mean value over all elements.

`minCoeff`(*(Vector6i)arg1*) → int :
Minimum value over all elements.

`outer`(*(Vector6i)arg1*, *(Vector6i)other*) → object :
Outer product with *other*.

`prod`(*(Vector6i)arg1*) → int :
Product of all elements.

`rows`(*(Vector6i)arg1*) → int :
Number of rows.

`sum`(*(Vector6i)arg1*) → int :
Sum of all elements.

`tail`(*(Vector6i)arg1*) → *Vector3i*

`class VectorX`
Dynamic-sized float vector.

Supported operations (`f` if a float/int, `v` is a VectorX): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`.

Implicit conversion from sequence (list, tuple, ...) of X floats.

`static Ones`(*(int)arg1*) → *HP2.VectorX*

`static Random`(*(int)len*) → HP2.VectorX :
Return vector of given length with all elements set to values between 0 and 1 randomly.

`static Unit`(*(int)arg1*, *(int)arg2*) → *HP2.VectorX*

`static Zero`(*(int)arg1*) → *HP2.VectorX*

`__init__`(*(object)arg1*) → None
___init___((object)arg1, (HP2.VectorX)other) -> None

___init___((object)arg1, (object)vv) -> object

`asDiagonal`(*(HP2.VectorX)arg1*) → HP2.MatrixX :
Return diagonal matrix with this vector on the diagonal.

`cols`(*(HP2.VectorX)arg1*) → int :
Number of columns.

`dot`(*(HP2.VectorX)arg1*, *(HP2.VectorX)other*) → HP2.Real :
Dot product with *other*.

`isApprox`(*(HP2.VectorX)arg1*, *(HP2.VectorX)other*[, *(HP2.Real)prec=Real("3.84273543930596175698289618669828536 4e-31")*]) → bool :
Approximate comparison with precision *prec*.

`maxAbsCoeff`(*(HP2.VectorX)arg1*) → HP2.Real :
Maximum absolute value over all elements.

`maxCoeff`(*(HP2.VectorX)arg1*) → HP2.Real :
Maximum value over all elements.

`mean`(*(HP2.VectorX)arg1*) → HP2.Real :
Mean value over all elements.

`minCoeff`(*(HP2.VectorX)arg1*) → HP2.Real :
Minimum value over all elements.

`norm`(*(HP2.VectorX)arg1*) → HP2.Real :
Euclidean norm.

`normalize`(*(HP2.VectorX)arg1*) → None :
Normalize this object in-place.

`normalized`(*(HP2.VectorX)arg1*) → HP2.VectorX :
Return normalized copy of this object

`outer`(*(HP2.VectorX)arg1*, *(HP2.VectorX)other*) → HP2.MatrixX :
Outer product with *other*.

`prod`(*(HP2.VectorX)arg1*) → HP2.Real :
Product of all elements.

`pruned`(*(HP2.VectorX)arg1*[, *(float)absTol=1e-06*]) → HP2.VectorX :
Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

`resize`(*(HP2.VectorX)arg1*, *(int)arg2*) → None

`rows`(*(HP2.VectorX)arg1*) → int :
Number of rows.

`squaredNorm`(*(HP2.VectorX)arg1*) → HP2.Real :
Square of the Euclidean norm.

`sum`(*(HP2.VectorX)arg1*) → HP2.Real :
Sum of all elements.

`class VectorXc`
/TODO/

`static Ones`(*(int)arg1*) → *HP2.VectorXc*

`static Random`(*(int)len*) → HP2.VectorXc :
Return vector of given length with all elements set to values between 0 and 1 randomly.

`static Unit`(*(int)arg1*, *(int)arg2*) → *HP2.VectorXc*

`static Zero`(*(int)arg1*) → *HP2.VectorXc*

__init__(*(object)arg1*) → None

___init___((object)arg1, (HP2.VectorXc)other) -> None

___init___((object)arg1, (object)vv) -> object

asDiagonal(*(HP2.VectorXc)arg1*) → HP2.MatrixXc :

Return diagonal matrix with this vector on the diagonal.

cols(*(HP2.VectorXc)arg1*) → int :

Number of columns.

dot(*(HP2.VectorXc)arg1*, *(HP2.VectorXc)other*) → HP2.Complex :

Dot product with *other*.

isApprox(*(HP2.VectorXc)arg1*, *(HP2.VectorXc)other*[, *(HP2.Real)prec=Complex("3.842735439305961756982896186698285364e-31", "0")*]) → bool :

Approximate comparison with precision *prec*.

maxAbsCoeff(*(HP2.VectorXc)arg1*) → HP2.Real :

Maximum absolute value over all elements.

mean(*(HP2.VectorXc)arg1*) → HP2.Complex :

Mean value over all elements.

norm(*(HP2.VectorXc)arg1*) → HP2.Real :

Euclidean norm.

normalize(*(HP2.VectorXc)arg1*) → None :

Normalize this object in-place.

normalized(*(HP2.VectorXc)arg1*) → HP2.VectorXc :

Return normalized copy of this object

outer(*(HP2.VectorXc)arg1*, *(HP2.VectorXc)other*) → HP2.MatrixXc :

Outer product with *other*.

prod(*(HP2.VectorXc)arg1*) → HP2.Complex :

Product of all elements.

pruned(*(HP2.VectorXc)arg1*[, *(float)absTol=1e-06*]) → HP2.VectorXc :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

resize(*(HP2.VectorXc)arg1*, *(int)arg2*) → None

rows(*(HP2.VectorXc)arg1*) → int :

Number of rows.

squaredNorm(*(HP2.VectorXc)arg1*) → HP2.Real :

Square of the Euclidean norm.

sum(*(HP2.VectorXc)arg1*) → HP2.Complex :

Sum of all elements.

vectorize = False

class yade._minieigenHP.**Matrix3**

3x3 float matrix.

Supported operations (`m` is a Matrix3, `f` if a float/int, `v` is a Vector3): `-m`, `m+m`, `m+=m`, `m-m`, `m-=m`, `m*f`, `f*m`, `m*=f`, `m/f`, `m/=f`, `m*m`, `m*=m`, `m*v`, `v*m`, `m==m`, `m!=m`.

Static attributes: `Zero`, `Ones`, `Identity`.

Identity = Matrix3(1,0,0, 0,1,0, 0,0,1)

Ones = Matrix3(1,1,1, 1,1,1, 1,1,1)

static Random() → Matrix3 :

Return an object where all elements are randomly set to values between 0 and 1.

Zero = Matrix3(0,0,0, 0,0,0, 0,0,0)

__init__(*(object)arg1*) → None

___init___((object)arg1, (Quaternion)q) -> None

___init___((object)arg1, (Matrix3)other) -> None

___init___((object)arg1, (Vector3)diag) -> object

___init___((object)arg1, (float)m00, (float)m01, (float)m02, (float)m10, (float)m11, (float)m12, (float)m20, (float)m21, (float)m22) -> object

___init___((object)arg1, (str)m00, (str)m01, (str)m02, (str)m10, (str)m11, (str)m12, (str)m20, (str)m21, (str)m22) -> object

___init___((object)arg1, (Vector3)r0, (Vector3)r1, (Vector3)r2 [, (bool)cols=False]) -> object

col(*(Matrix3)arg1*, *(int)col*) → Vector3 :

Return column as vector.

cols(*(Matrix3)arg1*) → int :

Number of columns.

computeUnitaryPositive(*(Matrix3)arg1*) → tuple :

Compute polar decomposition (unitary matrix U and positive semi-definite symmetric matrix P such that self=U*P).

determinant(*(Matrix3)arg1*) → float :

Return matrix determinant.

diagonal(*(Matrix3)arg1*) → Vector3 :

Return diagonal as vector.

inverse(*(Matrix3)arg1*) → Matrix3 :

Return inverted matrix.

isApprox(*(Matrix3)arg1*, *(Matrix3)other*[, *(float)prec=1e-12*]) → bool :

Approximate comparison with precision *prec*.

jacobiSVD(*(Matrix3)arg1*) → tuple :

Compute SVD decomposition of square matrix, retuns (U,S,V) such that self=U*S*V.transpose()

maxAbsCoeff(*(Matrix3)arg1*) → float :

Maximum absolute value over all elements.

maxCoeff(*(Matrix3)arg1*) → float :

Maximum value over all elements.

mean(*(Matrix3)arg1*) → float :

Mean value over all elements.

minCoeff(*(Matrix3)arg1*) → float :

Minimum value over all elements.

norm(*(Matrix3)arg1*) → float :

Euclidean norm.

normalize(*(Matrix3)arg1*) → None :

Normalize this object in-place.

normalized(*(Matrix3)arg1*) → Matrix3 :

Return normalized copy of this object

polarDecomposition(*(Matrix3)arg1*) → tuple :

Alias for *computeUnitaryPositive*.

prod(*(Matrix3)arg1*) → float :

Product of all elements.

pruned(*(Matrix3)arg1*[, *(float)absTol=1e-06*]) → Matrix3 :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

row(*(Matrix3)arg1*, *(int)row*) → Vector3 :

Return row as vector.

rows(*(Matrix3)arg1*) → int :

Number of rows.

selfAdjointEigenDecomposition(*(Matrix3)arg1*) → tuple :

Compute eigen (spectral) decomposition of symmetric matrix, returns (eigVecs,eigVals). eigVecs is orthogonal Matrix3 with columns ar normalized eigenvectors, eigVals is Vector3 with corresponding eigenvalues. self=eigVecs*diag(eigVals)*eigVecs.transpose().

spectralDecomposition(*(Matrix3)arg1*) → tuple :

Alias for *selfAdjointEigenDecomposition*.

squaredNorm(*(Matrix3)arg1*) → float :

Square of the Euclidean norm.

sum(*(Matrix3)arg1*) → float :

Sum of all elements.

svd(*(Matrix3)arg1*) → tuple :

Alias for *jacobiSVD*.

trace(*(Matrix3)arg1*) → float :

Return sum of diagonal elements.

transpose(*(Matrix3)arg1*) → Matrix3 :

Return transposed matrix.

class yade._minieigenHP.**Matrix3c**

/TODO/

Identity = Matrix3c(1,0,0, 0,1,0, 0,0,1)

Ones = Matrix3c(1,1,1, 1,1,1, 1,1,1)

static Random() → Matrix3c :

Return an object where all elements are randomly set to values between 0 and 1.

Zero = Matrix3c(0,0,0, 0,0,0, 0,0,0)

__init__(*(object)arg1*) → None

___init___((object)arg1, (Matrix3c)other) -> None

___init___((object)arg1, (Vector3c)diag) -> object

___init___((object)arg1, (complex)m00, (complex)m01, (complex)m02, (complex)m10, (complex)m11, (complex)m12, (complex)m20, (complex)m21, (complex)m22) -> object

___init___((object)arg1, (str)m00, (str)m01, (str)m02, (str)m10, (str)m11, (str)m12, (str)m20, (str)m21, (str)m22) -> object

___init___((object)arg1, (Vector3c)r0, (Vector3c)r1, (Vector3c)r2 [, (bool)cols=False]) -> object

col(*(Matrix3c)arg1*, *(int)col*) → Vector3c :

Return column as vector.

cols(*(Matrix3c)arg1*) → int :

Number of columns.

determinant(*(Matrix3c)arg1*) → complex :

Return matrix determinant.

diagonal(*(Matrix3c)arg1*) → Vector3c :

Return diagonal as vector.

inverse(*(Matrix3c)arg1*) → Matrix3c :

Return inverted matrix.

isApprox(*(Matrix3c)arg1*, *(Matrix3c)other*[, *(float)prec=1e-12*]) → bool :

Approximate comparison with precision *prec*.

maxAbsCoeff(*(Matrix3c)arg1*) → float :

Maximum absolute value over all elements.

mean(*(Matrix3c)arg1*) → complex :

Mean value over all elements.

norm(*(Matrix3c)arg1*) → float :

Euclidean norm.

normalize(*(Matrix3c)arg1*) → None :

Normalize this object in-place.

normalized(*(Matrix3c)arg1*) → Matrix3c :

Return normalized copy of this object

prod(*(Matrix3c)arg1*) → complex :

Product of all elements.

pruned(*(Matrix3c)arg1*[, *(float)absTol=1e-06*]) → Matrix3c :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

row(*(Matrix3c)arg1*, *(int)row*) → Vector3c :

Return row as vector.

rows(*(Matrix3c)arg1*) → int :

Number of rows.

squaredNorm(*(Matrix3c)arg1*) → float :

Square of the Euclidean norm.

sum(*(Matrix3c)arg1*) → complex :

Sum of all elements.

trace(*(Matrix3c)arg1*) → complex :

Return sum of diagonal elements.

transpose(*(Matrix3c)arg1*) → Matrix3c :

Return transposed matrix.

class yade._minieigenHP.Matrix6

6x6 float matrix. Constructed from 4 3x3 sub-matrices, from 6xVector6 (rows).

Supported operations (m is a Matrix6, f if a float/int, v is a Vector6): -m, m+m, m+=m, m-m, m-=m, m*f, f*m, m*=f, m/f, m/=f, m*m, m*=m, m*v, v*m, m==m, m!=m.

Static attributes: Zero, Ones, Identity.

Identity = Matrix6((1,0,0,0,0,0), (0,1,0,0,0,0), (0,0,1,0,0,0), (0,0,0,1,0,0), (0,0,0,0,1,0), (0,0,0,0,0,1))

Ones = Matrix6((1,1,1,1,1,1), (1,1,1,1,1,1), (1,1,1,1,1,1), (1,1,1,1,1,1), (1,1,1,1,1,1), (1,1,1,1,1,1))

static Random() → Matrix6 :

Return an object where all elements are randomly set to values between 0 and 1.

Zero = Matrix6((0,0,0,0,0,0), (0,0,0,0,0,0), (0,0,0,0,0,0), (0,0,0,0,0,0), (0,0,0,0,0,0), (0,0,0,0,0,0))

__init__(*(object)arg1*) → None

___init___((object)arg1, (Matrix6)other) -> None

___init___((object)arg1, (Vector6)diag) -> object

___init___((object)arg1, (Matrix3)ul, (Matrix3)ur, (Matrix3)ll, (Matrix3)lr) -> object

___init___((object)arg1, (Vector6)l0, (Vector6)l1, (Vector6)l2, (Vector6)l3, (Vector6)l4, (Vector6)l5 [, (bool)cols=False]) -> object

col(*(Matrix6)arg1*, *(int)col*) → Vector6 :

Return column as vector.

cols(*(Matrix6)arg1*) → int :

Number of columns.

computeUnitaryPositive(*(Matrix6)arg1*) → tuple :

Compute polar decomposition (unitary matrix U and positive semi-definite symmetric matrix P such that self=U*P).

determinant(*(Matrix6)arg1*) → float :

Return matrix determinant.

diagonal(*(Matrix6)arg1*) → Vector6 :

Return diagonal as vector.

inverse(*(Matrix6)arg1*) → Matrix6 :

Return inverted matrix.

isApprox(*(Matrix6)arg1*, *(Matrix6)other*[, *(float)prec=1e-12*]) → bool :

Approximate comparison with precision *prec*.

jacobiSVD(*(Matrix6)arg1*) → tuple :

Compute SVD decomposition of square matrix, retuns (U,S,V) such that self=U*S*V.transpose()

ll(*(Matrix6)arg1*) → Matrix3 :

Return lower-left 3x3 block

lr(*(Matrix6)arg1*) → Matrix3 :

Return lower-right 3x3 block

maxAbsCoeff(*(Matrix6)arg1*) → float :

Maximum absolute value over all elements.

maxCoeff(*(Matrix6)arg1*) → float :

Maximum value over all elements.

mean(*(Matrix6)arg1*) → float :

Mean value over all elements.

minCoeff(*(Matrix6)arg1*) → float :

Minimum value over all elements.

norm(*(Matrix6)arg1*) → float :

Euclidean norm.

normalize(*(Matrix6)arg1*) → None :

Normalize this object in-place.

normalized(*(Matrix6)arg1*) → Matrix6 :

Return normalized copy of this object

polarDecomposition(*(Matrix6)arg1*) → tuple :

Alias for *computeUnitaryPositive*.

prod(*(Matrix6)arg1*) → float :

Product of all elements.

pruned(*(Matrix6)arg1*[, *(float)absTol=1e-06*]) → Matrix6 :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

row(*(Matrix6)arg1*, *(int)row*) → Vector6 :

Return row as vector.

rows(*(Matrix6)arg1*) → int :

Number of rows.

selfAdjointEigenDecomposition(*(Matrix6)arg1*) → tuple :

Compute eigen (spectral) decomposition of symmetric matrix, returns (eigVecs,eigVals). eigVecs is orthogonal Matrix3 with columns ar normalized eigenvectors, eigVals is Vector3 with corresponding eigenvalues. self=eigVecs*diag(eigVals)*eigVecs.transpose().

spectralDecomposition(*(Matrix6)arg1*) → tuple :

Alias for *selfAdjointEigenDecomposition*.

squaredNorm(*(Matrix6)arg1*) → float :

Square of the Euclidean norm.

sum(*(Matrix6)arg1*) → float :

Sum of all elements.

svd(*(Matrix6)arg1*) → tuple :

Alias for *jacobiSVD*.

trace(*(Matrix6)arg1*) → float :

Return sum of diagonal elements.

transpose(*(Matrix6)arg1*) → Matrix6 :

Return transposed matrix.

ul(*(Matrix6)arg1*) → Matrix3 :

Return upper-left 3x3 block

ur(*(Matrix6)arg1*) → Matrix3 :

Return upper-right 3x3 block

class yade._minieigenHP.Matrix6c

/TODO/

Identity = Matrix6c((1,0,0,0,0,0), (0,1,0,0,0,0), (0,0,1,0,0,0), (0,0,0,1,0,0), (0,0,0,0,1,0), (0,0,0,0,0,1))

Ones = Matrix6c((1,1,1,1,1,1), (1,1,1,1,1,1), (1,1,1,1,1,1), (1,1,1,1,1,1), (1,1,1,1,1,1), (1,1,1,1,1,1))

static Random() → Matrix6c :

Return an object where all elements are randomly set to values between 0 and 1.

Zero = Matrix6c((0,0,0,0,0,0), (0,0,0,0,0,0), (0,0,0,0,0,0), (0,0,0,0,0,0), (0,0,0,0,0,0), (0,0,0,0,0,0))

__init__(*(object)arg1***)** → None

___init___((object)arg1, (Matrix6c)other) -> None

___init___((object)arg1, (Vector6c)diag) -> object

___init___((object)arg1, (Matrix3c)ul, (Matrix3c)ur, (Matrix3c)ll, (Matrix3c)lr) -> object

___init___((object)arg1, (Vector6c)l0, (Vector6c)l1, (Vector6c)l2, (Vector6c)l3, (Vector6c)l4, (Vector6c)l5 [, (bool)cols=False]) -> object

col(*(Matrix6c)arg1***,** *(int)col***)** → Vector6c :

Return column as vector.

cols(*(Matrix6c)arg1***)** → int :

Number of columns.

determinant(*(Matrix6c)arg1***)** → complex :

Return matrix determinant.

diagonal(*(Matrix6c)arg1***)** → Vector6c :

Return diagonal as vector.

inverse(*(Matrix6c)arg1***)** → Matrix6c :

Return inverted matrix.

isApprox(*(Matrix6c)arg1***,** *(Matrix6c)other*[**,** *(float)prec=1e-12*]**)** → bool :

Approximate comparison with precision *prec*.

ll(*(Matrix6c)arg1***)** → Matrix3c :

Return lower-left 3x3 block

lr(*(Matrix6c)arg1***)** → Matrix3c :

Return lower-right 3x3 block

maxAbsCoeff(*(Matrix6c)arg1***)** → float :

Maximum absolute value over all elements.

mean(*(Matrix6c)arg1***)** → complex :

Mean value over all elements.

norm(*(Matrix6c)arg1***)** → float :

Euclidean norm.

normalize(*(Matrix6c)arg1***)** → None :

Normalize this object in-place.

normalized(*(Matrix6c)arg1***)** → Matrix6c :

Return normalized copy of this object

prod(*(Matrix6c)arg1*) → complex :

Product of all elements.

pruned(*(Matrix6c)arg1*[, *(float)absTol=1e-06*]) → Matrix6c :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

row(*(Matrix6c)arg1*, *(int)row*) → Vector6c :

Return row as vector.

rows(*(Matrix6c)arg1*) → int :

Number of rows.

squaredNorm(*(Matrix6c)arg1*) → float :

Square of the Euclidean norm.

sum(*(Matrix6c)arg1*) → complex :

Sum of all elements.

trace(*(Matrix6c)arg1*) → complex :

Return sum of diagonal elements.

transpose(*(Matrix6c)arg1*) → Matrix6c :

Return transposed matrix.

ul(*(Matrix6c)arg1*) → Matrix3c :

Return upper-left 3x3 block

ur(*(Matrix6c)arg1*) → Matrix3c :

Return upper-right 3x3 block

class yade._minieigenHP.**MatrixX**

XxX (dynamic-sized) float matrix. Constructed from list of rows (as VectorX).

Supported operations (`m` is a MatrixX, `f` if a float/int, `v` is a VectorX): `-m`, `m+m`, `m+=m`, `m-m`, `m-=m`, `m*f`, `f*m`, `m*=f`, `m/f`, `m/=f`, `m*m`, `m*=m`, `m*v`, `v*m`, `m==m`, `m!=m`.

static Identity(*(int)arg1*, *(int)rank*) → MatrixX :

Create identity matrix with given rank (square).

static Ones(*(int)rows*, *(int)cols*) → MatrixX :

Create matrix of given dimensions where all elements are set to 1.

static Random(*(int)rows*, *(int)cols*) → MatrixX :

Create matrix with given dimensions where all elements are set to number between 0 and 1 (uniformly-distributed).

static Zero(*(int)rows*, *(int)cols*) → MatrixX :

Create zero matrix of given dimensions

__init__(*(object)arg1*) → None

___init___((object)arg1, (MatrixX)other) -> None

___init___((object)arg1, (VectorX)diag) -> object

___init___((object)arg1 [, (VectorX)r0=VectorX() [, (VectorX)r1=VectorX() [, (VectorX)r2=VectorX() [, (VectorX)r3=VectorX() [, (VectorX)r4=VectorX() [, (VectorX)r5=VectorX() [, (VectorX)r6=VectorX() [, (VectorX)r7=VectorX() [, (VectorX)r8=VectorX() [, (VectorX)r9=VectorX() [, (bool)cols=False]]]]]]]]]]) -> object

___init___((object)arg1, (object)rows [, (bool)cols=False]) -> object

col(*(MatrixX)arg1*, *(int)col*) → VectorX :

Return column as vector.

cols(*(MatrixX)arg1*) → int :

Number of columns.

computeUnitaryPositive(*(MatrixX)arg1*) → tuple :

Compute polar decomposition (unitary matrix U and positive semi-definite symmetric matrix P such that self=U*P).

determinant(*(MatrixX)arg1*) → float :

Return matrix determinant.

diagonal(*(MatrixX)arg1*) → VectorX :

Return diagonal as vector.

inverse(*(MatrixX)arg1*) → MatrixX :

Return inverted matrix.

isApprox(*(MatrixX)arg1*, *(MatrixX)other*[, *(float)prec=1e-12*]) → bool :

Approximate comparison with precision *prec*.

jacobiSVD(*(MatrixX)arg1*) → tuple :

Compute SVD decomposition of square matrix, retuns (U,S,V) such that self=U*S*V.transpose()

maxAbsCoeff(*(MatrixX)arg1*) → float :

Maximum absolute value over all elements.

maxCoeff(*(MatrixX)arg1*) → float :

Maximum value over all elements.

mean(*(MatrixX)arg1*) → float :

Mean value over all elements.

minCoeff(*(MatrixX)arg1*) → float :

Minimum value over all elements.

norm(*(MatrixX)arg1*) → float :

Euclidean norm.

normalize(*(MatrixX)arg1*) → None :

Normalize this object in-place.

normalized(*(MatrixX)arg1*) → MatrixX :

Return normalized copy of this object

polarDecomposition(*(MatrixX)arg1*) → tuple :

Alias for *computeUnitaryPositive*.

prod(*(MatrixX)arg1*) → float :

Product of all elements.

pruned(*(MatrixX)arg1*[, *(float)absTol=1e-06*]) → MatrixX :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

resize(*(MatrixX)arg1*, *(int)rows*, *(int)cols*) → None :

Change size of the matrix, keep values of elements which exist in the new matrix

row(*(MatrixX)arg1*, *(int)row*) → VectorX :

Return row as vector.

rows(*(MatrixX)arg1*) → int :

Number of rows.

selfAdjointEigenDecomposition(*(MatrixX)arg1*) → tuple :

Compute eigen (spectral) decomposition of symmetric matrix, returns (eigVecs,eigVals). eigVecs is orthogonal Matrix3 with columns ar normalized eigenvectors, eigVals is Vector3 with corresponding eigenvalues. self=eigVecs*diag(eigVals)*eigVecs.transpose().

spectralDecomposition(*(MatrixX)arg1*) → tuple :

Alias for *selfAdjointEigenDecomposition*.

squaredNorm(*(MatrixX)arg1*) → float :

Square of the Euclidean norm.

sum(*(MatrixX)arg1*) → float :

Sum of all elements.

svd(*(MatrixX)arg1*) → tuple :

Alias for *jacobiSVD*.

trace(*(MatrixX)arg1*) → float :

Return sum of diagonal elements.

transpose(*(MatrixX)arg1*) → MatrixX :

Return transposed matrix.

class yade._minieigenHP.**MatrixXc**

/TODO/

static Identity(*(int)arg1*, *(int)rank*) → MatrixXc :

Create identity matrix with given rank (square).

static Ones(*(int)rows*, *(int)cols*) → MatrixXc :

Create matrix of given dimensions where all elements are set to 1.

static Random(*(int)rows*, *(int)cols*) → MatrixXc :

Create matrix with given dimensions where all elements are set to number between 0 and 1 (uniformly-distributed).

static Zero(*(int)rows*, *(int)cols*) → MatrixXc :

Create zero matrix of given dimensions

__init__(*(object)arg1*) → None

___init___((object)arg1, (MatrixXc)other) -> None

___init___((object)arg1, (VectorXc)diag) -> object

___init___((object)arg1 [, (VectorXc)r0=VectorXc() [, (VectorXc)r1=VectorXc() [, (VectorXc)r2=VectorXc() [, (VectorXc)r3=VectorXc() [, (VectorXc)r4=VectorXc() [, (VectorXc)r5=VectorXc() [, (VectorXc)r6=VectorXc() [, (VectorXc)r7=VectorXc() [, (VectorXc)r8=VectorXc() [, (VectorXc)r9=VectorXc() [, (bool)cols=False]]]]]]]]]]) -> object

___init___((object)arg1, (object)rows [, (bool)cols=False]) -> object

col(*(MatrixXc)arg1*, *(int)col*) → VectorXc :

Return column as vector.

cols(*(MatrixXc)arg1*) → int :

Number of columns.

determinant(*(MatrixXc)arg1*) → complex :

Return matrix determinant.

diagonal(*(MatrixXc)arg1*) → VectorXc :

Return diagonal as vector.

inverse(*(MatrixXc)arg1*) → MatrixXc :
Return inverted matrix.

isApprox(*(MatrixXc)arg1*, *(MatrixXc)other*[, *(float)prec=1e-12*]) → bool :
Approximate comparison with precision *prec*.

maxAbsCoeff(*(MatrixXc)arg1*) → float :
Maximum absolute value over all elements.

mean(*(MatrixXc)arg1*) → complex :
Mean value over all elements.

norm(*(MatrixXc)arg1*) → float :
Euclidean norm.

normalize(*(MatrixXc)arg1*) → None :
Normalize this object in-place.

normalized(*(MatrixXc)arg1*) → MatrixXc :
Return normalized copy of this object

prod(*(MatrixXc)arg1*) → complex :
Product of all elements.

pruned(*(MatrixXc)arg1*[, *(float)absTol=1e-06*]) → MatrixXc :
Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

resize(*(MatrixXc)arg1*, *(int)rows*, *(int)cols*) → None :
Change size of the matrix, keep values of elements which exist in the new matrix

row(*(MatrixXc)arg1*, *(int)row*) → VectorXc :
Return row as vector.

rows(*(MatrixXc)arg1*) → int :
Number of rows.

squaredNorm(*(MatrixXc)arg1*) → float :
Square of the Euclidean norm.

sum(*(MatrixXc)arg1*) → complex :
Sum of all elements.

trace(*(MatrixXc)arg1*) → complex :
Return sum of diagonal elements.

transpose(*(MatrixXc)arg1*) → MatrixXc :
Return transposed matrix.

class yade._minieigenHP.**Quaternion**
Quaternion representing rotation.

Supported operations (`q` is a Quaternion, `v` is a Vector3): `q*q` (rotation composition), `q*=q`, `q*v` (rotating `v` by `q`), `q==q`, `q!=q`.

Static attributes: `Identity`.

Note

Quaternion is represented as axis-angle when printed (e.g. `Identity` is `Quaternion((1,0,0), 0)`, and can also be constructed from the axis-angle representation. This is however different from the data stored inside, which can be accessed by indices `[0]` (x), `[1]` (y), `[2]` (z), `[3]` (w). To obtain axis-angle programatically, use *`Quaternion.toAxisAngle`* which returns the tuple.

Identity = Quaternion((1,0,0),0)

Rotate(*(Quaternion)arg1*, *(Vector3)v*) → *Vector3*

__init__(*(object)arg1*) → None

___init___((object)arg1, (Vector3)axis, (object)angle) -> object

___init___((object)arg1, (Vector3)axis, (float)angle) -> object

___init___((object)arg1, (object)angle, (Vector3)axis) -> object

___init___((object)arg1, (float)angle, (Vector3)axis) -> object

___init___((object)arg1, (tuple)axis, (str)angle) -> object

___init___((object)arg1, (tuple)tuple) -> object

___init___((object)arg1, (Vector3)u, (Vector3)v) -> object

___init___((object)arg1, (str)str1, (str)str2, (str)str3, (str)str4) -> object

___init___((object)arg1, (float)w, (float)x, (float)y, (float)z) -> None :
Initialize from coefficients.

> **Note**
>
> The order of coefficients is *w*, *x*, *y*, *z*. The [] operator numbers them differently, 0...4 for *x y z w*!

___init___((object)arg1, (Matrix3)rotMatrix) -> None

___init___((object)arg1, (Quaternion)other) -> None

angularDistance(*(Quaternion)arg1*, *(Quaternion)arg2*) → float

conjugate(*(Quaternion)arg1*) → *Quaternion*

inverse(*(Quaternion)arg1*) → *Quaternion*

norm(*(Quaternion)arg1*) → float

normalize(*(Quaternion)arg1*) → None

normalized(*(Quaternion)arg1*) → *Quaternion*

setFromTwoVectors(*(Quaternion)arg1*, *(Vector3)u*, *(Vector3)v*) → None

slerp(*(Quaternion)arg1*, *(float)t*, *(Quaternion)other*) → *Quaternion*

toAngleAxis(*(Quaternion)arg1*) → tuple

toAxisAngle(*(Quaternion)arg1*) → tuple

toRotationMatrix(*(Quaternion)arg1*) → *Matrix3*

toRotationVector(*(Quaternion)arg1*) → *Vector3*

class yade._minieigenHP.**Vector2**

3-dimensional float vector.

Supported operations (`f` if a float/int, `v` is a Vector3): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`.

Implicit conversion from sequence (list, tuple, ...) of 2 floats.

Static attributes: `Zero`, `Ones`, `UnitX`, `UnitY`.

Identity = Vector2(1,0)

Ones = Vector2(1,1)

static Random() → Vector2 :

Return an object where all elements are randomly set to values between 0 and 1.

static Unit(*(int)arg1***)** → *Vector2*

UnitX = Vector2(1,0)

UnitY = Vector2(0,1)

Zero = Vector2(0,0)

__init__(*(object)arg1***)** → None

___init___((object)arg1, (Vector2)other) -> None

___init___((object)arg1, (str)str1, (str)str2) -> object

___init___((object)arg1, (float)x, (float)y) -> None

asDiagonal(*(Vector2)arg1***)** → object :

Return diagonal matrix with this vector on the diagonal.

cols(*(Vector2)arg1***)** → int :

Number of columns.

dot(*(Vector2)arg1*, *(Vector2)other***)** → float :

Dot product with *other*.

isApprox(*(Vector2)arg1*, *(Vector2)other*[, *(float)prec=1e-12*]**)** → bool :

Approximate comparison with precision *prec*.

maxAbsCoeff(*(Vector2)arg1***)** → float :

Maximum absolute value over all elements.

maxCoeff(*(Vector2)arg1***)** → float :

Maximum value over all elements.

mean(*(Vector2)arg1***)** → float :

Mean value over all elements.

minCoeff(*(Vector2)arg1***)** → float :

Minimum value over all elements.

norm(*(Vector2)arg1***)** → float :

Euclidean norm.

normalize(*(Vector2)arg1***)** → None :

Normalize this object in-place.

normalized(*(Vector2)arg1***)** → Vector2 :

Return normalized copy of this object

outer(*(Vector2)arg1*, *(Vector2)other***)** → object :

Outer product with *other*.

prod(*(Vector2)arg1***)** → float :

Product of all elements.

pruned(*(Vector2)arg1*[, *(float)absTol=1e-06*]**)** → Vector2 :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

rows(*(Vector2)arg1*) → int :
Number of rows.

squaredNorm(*(Vector2)arg1*) → float :
Square of the Euclidean norm.

sum(*(Vector2)arg1*) → float :
Sum of all elements.

class yade._minieigenHP.**Vector2c**

/TODO/

Identity = Vector2c(1,0)

Ones = Vector2c(1,1)

static Random() → Vector2c :
Return an object where all elements are randomly set to values between 0 and 1.

static Unit(*(int)arg1*) → *Vector2c*

UnitX = Vector2c(1,0)

UnitY = Vector2c(0,1)

Zero = Vector2c(0,0)

__init__(*(object)arg1*) → None
___init___((object)arg1, (Vector2c)other) -> None
___init___((object)arg1, (str)str1, (str)str2) -> object
___init___((object)arg1, (complex)x, (complex)y) -> None

asDiagonal(*(Vector2c)arg1*) → object :
Return diagonal matrix with this vector on the diagonal.

cols(*(Vector2c)arg1*) → int :
Number of columns.

dot(*(Vector2c)arg1*, *(Vector2c)other*) → complex :
Dot product with *other*.

isApprox(*(Vector2c)arg1*, *(Vector2c)other*[, *(float)prec=1e-12*]) → bool :
Approximate comparison with precision *prec*.

maxAbsCoeff(*(Vector2c)arg1*) → float :
Maximum absolute value over all elements.

mean(*(Vector2c)arg1*) → complex :
Mean value over all elements.

norm(*(Vector2c)arg1*) → float :
Euclidean norm.

normalize(*(Vector2c)arg1*) → None :
Normalize this object in-place.

normalized(*(Vector2c)arg1*) → Vector2c :
Return normalized copy of this object

outer(*(Vector2c)arg1*, *(Vector2c)other*) → object :
Outer product with *other*.

prod(*(Vector2c)arg1*) → complex :

Product of all elements.

pruned(*(Vector2c)arg1*[, *(float)absTol=1e-06*]) → Vector2c :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

rows(*(Vector2c)arg1*) → int :

Number of rows.

squaredNorm(*(Vector2c)arg1*) → float :

Square of the Euclidean norm.

sum(*(Vector2c)arg1*) → complex :

Sum of all elements.

class yade._minieigenHP.Vector2i

2-dimensional integer vector.

Supported operations (`i` if an int, `v` is a Vector2i): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*i`, `i*v`, `v*=i`, `v==v`, `v!=v`.

Implicit conversion from sequence (list, tuple, …) of 2 integers.

Static attributes: **Zero**, **Ones**, **UnitX**, **UnitY**.

Identity = Vector2i(1,0)

Ones = Vector2i(1,1)

static Random() → Vector2i :

Return an object where all elements are randomly set to values between 0 and 1.

static Unit(*(int)arg1*) → *Vector2i*

UnitX = Vector2i(1,0)

UnitY = Vector2i(0,1)

Zero = Vector2i(0,0)

__init__(*(object)arg1*) → None

___init___((object)arg1, (Vector2i)other) -> None

___init___((object)arg1, (str)str1, (str)str2) -> object

___init___((object)arg1, (int)x, (int)y) -> None

asDiagonal(*(Vector2i)arg1*) → object :

Return diagonal matrix with this vector on the diagonal.

cols(*(Vector2i)arg1*) → int :

Number of columns.

dot(*(Vector2i)arg1*, *(Vector2i)other*) → int :

Dot product with *other*.

isApprox(*(Vector2i)arg1*, *(Vector2i)other*[, *(int)prec=0*]) → bool :

Approximate comparison with precision *prec*.

maxAbsCoeff(*(Vector2i)arg1*) → int :

Maximum absolute value over all elements.

maxCoeff(*(Vector2i)arg1*) → int :

Maximum value over all elements.

mean(*(Vector2i)arg1*) → int :

Mean value over all elements.

minCoeff(*(Vector2i)arg1*) → int :

Minimum value over all elements.

outer(*(Vector2i)arg1*, *(Vector2i)other*) → object :

Outer product with *other*.

prod(*(Vector2i)arg1*) → int :

Product of all elements.

rows(*(Vector2i)arg1*) → int :

Number of rows.

sum(*(Vector2i)arg1*) → int :

Sum of all elements.

class yade._minieigenHP.Vector3

3-dimensional float vector.

Supported operations (`f` if a float/int, `v` is a Vector3): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`, plus operations with `Matrix3` and `Quaternion`.

Implicit conversion from sequence (list, tuple, ...) of 3 floats.

Static attributes: `Zero`, `Ones`, `UnitX`, `UnitY`, `UnitZ`.

Identity = Vector3(1,0,0)

Ones = Vector3(1,1,1)

static Random() → Vector3 :

Return an object where all elements are randomly set to values between 0 and 1.

static Unit(*(int)arg1*) → *Vector3*

UnitX = Vector3(1,0,0)

UnitY = Vector3(0,1,0)

UnitZ = Vector3(0,0,1)

Zero = Vector3(0,0,0)

__init__(*(object)arg1*) → None

___init___((object)arg1, (Vector3)other) -> None

___init___((object)arg1, (str)str1, (str)str2, (str)str3) -> object

___init___((object)arg1 [, (float)x=0.0 [, (float)y=0.0 [, (float)z=0.0]]]) -> None

asDiagonal(*(Vector3)arg1*) → Matrix3 :

Return diagonal matrix with this vector on the diagonal.

cols(*(Vector3)arg1*) → int :

Number of columns.

cross(*(Vector3)arg1*, *(Vector3)arg2*) → *Vector3*

dot(*(Vector3)arg1*, *(Vector3)other*) → float :

Dot product with *other*.

isApprox(*(Vector3)arg1*, *(Vector3)other*[, *(float)prec=1e-12*]) → bool :

Approximate comparison with precision *prec*.

maxAbsCoeff(*(Vector3)arg1*) → float :
Maximum absolute value over all elements.

maxCoeff(*(Vector3)arg1*) → float :
Maximum value over all elements.

mean(*(Vector3)arg1*) → float :
Mean value over all elements.

minCoeff(*(Vector3)arg1*) → float :
Minimum value over all elements.

norm(*(Vector3)arg1*) → float :
Euclidean norm.

normalize(*(Vector3)arg1*) → None :
Normalize this object in-place.

normalized(*(Vector3)arg1*) → Vector3 :
Return normalized copy of this object

outer(*(Vector3)arg1*, *(Vector3)other*) → Matrix3 :
Outer product with *other*.

prod(*(Vector3)arg1*) → float :
Product of all elements.

pruned(*(Vector3)arg1*[, *(float)absTol=1e-06*]) → Vector3 :
Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

rows(*(Vector3)arg1*) → int :
Number of rows.

squaredNorm(*(Vector3)arg1*) → float :
Square of the Euclidean norm.

sum(*(Vector3)arg1*) → float :
Sum of all elements.

xy(*(Vector3)arg1*) → *Vector2*

xz(*(Vector3)arg1*) → *Vector2*

yx(*(Vector3)arg1*) → *Vector2*

yz(*(Vector3)arg1*) → *Vector2*

zx(*(Vector3)arg1*) → *Vector2*

zy(*(Vector3)arg1*) → *Vector2*

class yade._minieigenHP.**Vector3c**

/TODO/

Identity = Vector3c(1,0,0)

Ones = Vector3c(1,1,1)

static Random() → Vector3c :
Return an object where all elements are randomly set to values between 0 and 1.

static Unit(*(int)arg1*) → *Vector3c*

UnitX = Vector3c(1,0,0)

UnitY = Vector3c(0,1,0)

UnitZ = Vector3c(0,0,1)

Zero = Vector3c(0,0,0)

__init__(*(object)arg1*) → None
___init___((object)arg1, (Vector3c)other) -> None
___init___((object)arg1, (str)str1, (str)str2, (str)str3) -> object
___init___((object)arg1 [, (complex)x=0j [, (complex)y=0j [, (complex)z=0j]]]) -> None

asDiagonal(*(Vector3c)arg1*) → Matrix3c :
Return diagonal matrix with this vector on the diagonal.

cols(*(Vector3c)arg1*) → int :
Number of columns.

cross(*(Vector3c)arg1*, *(Vector3c)arg2*) → *Vector3c*

dot(*(Vector3c)arg1*, *(Vector3c)other*) → complex :
Dot product with *other*.

isApprox(*(Vector3c)arg1*, *(Vector3c)other*[, *(float)prec=1e-12*]) → bool :
Approximate comparison with precision *prec*.

maxAbsCoeff(*(Vector3c)arg1*) → float :
Maximum absolute value over all elements.

mean(*(Vector3c)arg1*) → complex :
Mean value over all elements.

norm(*(Vector3c)arg1*) → float :
Euclidean norm.

normalize(*(Vector3c)arg1*) → None :
Normalize this object in-place.

normalized(*(Vector3c)arg1*) → Vector3c :
Return normalized copy of this object

outer(*(Vector3c)arg1*, *(Vector3c)other*) → Matrix3c :
Outer product with *other*.

prod(*(Vector3c)arg1*) → complex :
Product of all elements.

pruned(*(Vector3c)arg1*[, *(float)absTol=1e-06*]) → Vector3c :
Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

rows(*(Vector3c)arg1*) → int :
Number of rows.

squaredNorm(*(Vector3c)arg1*) → float :
Square of the Euclidean norm.

sum(*(Vector3c)arg1*) → complex :
Sum of all elements.

xy(*(Vector3c)arg1*) → *Vector2c*

xz(*(Vector3c)arg1*) → *Vector2c*

yx(*(Vector3c)arg1*) → *Vector2c*

yz(*(Vector3c)arg1*) → *Vector2c*

zx(*(Vector3c)arg1*) → *Vector2c*

zy(*(Vector3c)arg1*) → *Vector2c*

class yade._minieigenHP.**Vector3i**

3-dimensional integer vector.

Supported operations (`i` if an int, `v` is a Vector3i): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*i`, `i*v`, `v*=i`, `v==v`, `v!=v`.

Implicit conversion from sequence (list, tuple, …) of 3 integers.

Static attributes: `Zero`, `Ones`, `UnitX`, `UnitY`, `UnitZ`.

Identity = Vector3i(1,0,0)

Ones = Vector3i(1,1,1)

static Random() → Vector3i :

Return an object where all elements are randomly set to values between 0 and 1.

static Unit(*(int)arg1*) → *Vector3i*

UnitX = Vector3i(1,0,0)

UnitY = Vector3i(0,1,0)

UnitZ = Vector3i(0,0,1)

Zero = Vector3i(0,0,0)

__init__(*(object)arg1*) → None

___init___((object)arg1, (Vector3i)other) -> None

___init___((object)arg1, (str)str1, (str)str2, (str)str3) -> object

___init___((object)arg1 [, (int)x=0 [, (int)y=0 [, (int)z=0]]]) -> None

asDiagonal(*(Vector3i)arg1*) → object :

Return diagonal matrix with this vector on the diagonal.

cols(*(Vector3i)arg1*) → int :

Number of columns.

cross(*(Vector3i)arg1*, *(Vector3i)arg2*) → *Vector3i*

dot(*(Vector3i)arg1*, *(Vector3i)other*) → int :

Dot product with *other*.

isApprox(*(Vector3i)arg1*, *(Vector3i)other*[, *(int)prec=0*]) → bool :

Approximate comparison with precision *prec*.

maxAbsCoeff(*(Vector3i)arg1*) → int :

Maximum absolute value over all elements.

maxCoeff(*(Vector3i)arg1*) → int :

Maximum value over all elements.

mean(*(Vector3i)arg1*) → int :

Mean value over all elements.

minCoeff(*(Vector3i)arg1*) → int :

Minimum value over all elements.

outer(*(Vector3i)arg1*, *(Vector3i)other*) → object :

Outer product with *other*.

prod(*(Vector3i)arg1*) → int :

Product of all elements.

rows(*(Vector3i)arg1*) → int :

Number of rows.

sum(*(Vector3i)arg1*) → int :

Sum of all elements.

xy(*(Vector3i)arg1*) → *Vector2i*

xz(*(Vector3i)arg1*) → *Vector2i*

yx(*(Vector3i)arg1*) → *Vector2i*

yz(*(Vector3i)arg1*) → *Vector2i*

zx(*(Vector3i)arg1*) → *Vector2i*

zy(*(Vector3i)arg1*) → *Vector2i*

class yade._minieigenHP.**Vector4**

4-dimensional float vector.

Supported operations (`f` if a float/int, `v` is a Vector3): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`.

Implicit conversion from sequence (list, tuple, ...) of 4 floats.

Static attributes: `Zero`, `Ones`.

Identity = Vector4(1,0,0, 0)

Ones = Vector4(1,1,1, 1)

static Random() → Vector4 :

Return an object where all elements are randomly set to values between 0 and 1.

static Unit(*(int)arg1*) → *Vector4*

Zero = Vector4(0,0,0, 0)

__init__(*(object)arg1*) → None

___init___((object)arg1, (Vector4)other) -> None

___init___((object)arg1, (str)str1, (str)str2, (str)str3, (str)str4) -> object

___init___((object)arg1, (float)v0, (float)v1, (float)v2, (float)v3) -> None

asDiagonal(*(Vector4)arg1*) → object :

Return diagonal matrix with this vector on the diagonal.

cols(*(Vector4)arg1*) → int :

Number of columns.

dot(*(Vector4)arg1*, *(Vector4)other*) → float :

Dot product with *other*.

isApprox(*(Vector4)arg1*, *(Vector4)other*[, *(float)prec=1e-12*]) → bool :
Approximate comparison with precision *prec*.

maxAbsCoeff(*(Vector4)arg1*) → float :
Maximum absolute value over all elements.

maxCoeff(*(Vector4)arg1*) → float :
Maximum value over all elements.

mean(*(Vector4)arg1*) → float :
Mean value over all elements.

minCoeff(*(Vector4)arg1*) → float :
Minimum value over all elements.

norm(*(Vector4)arg1*) → float :
Euclidean norm.

normalize(*(Vector4)arg1*) → None :
Normalize this object in-place.

normalized(*(Vector4)arg1*) → Vector4 :
Return normalized copy of this object

outer(*(Vector4)arg1*, *(Vector4)other*) → object :
Outer product with *other*.

prod(*(Vector4)arg1*) → float :
Product of all elements.

pruned(*(Vector4)arg1*[, *(float)absTol=1e-06*]) → Vector4 :
Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

rows(*(Vector4)arg1*) → int :
Number of rows.

squaredNorm(*(Vector4)arg1*) → float :
Square of the Euclidean norm.

sum(*(Vector4)arg1*) → float :
Sum of all elements.

class yade._minieigenHP.**Vector6**
6-dimensional float vector.

Supported operations (`f` if a float/int, `v` is a Vector6): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`.

Implicit conversion from sequence (list, tuple, …) of 6 floats.

Static attributes: `Zero`, `Ones`.

Identity = Vector6(1,0,0, 0,0,0)

Ones = Vector6(1,1,1, 1,1,1)

static Random() → Vector6 :
Return an object where all elements are randomly set to values between 0 and 1.

static Unit(*(int)arg1*) → *Vector6*

Zero = Vector6(0,0,0, 0,0,0)

__init__(*(object)arg1*) → None

___init___((object)arg1, (Vector6)other) -> None

___init___((object)arg1, (float)v0, (float)v1, (float)v2, (float)v3, (float)v4, (float)v5) -> object

___init___((object)arg1, (Vector3)head, (Vector3)tail) -> object

asDiagonal(*(Vector6)arg1*) → Matrix6 :

Return diagonal matrix with this vector on the diagonal.

cols(*(Vector6)arg1*) → int :

Number of columns.

dot(*(Vector6)arg1*, *(Vector6)other*) → float :

Dot product with *other*.

head(*(Vector6)arg1*) → *Vector3*

isApprox(*(Vector6)arg1*, *(Vector6)other*[, *(float)prec=1e-12*]) → bool :

Approximate comparison with precision *prec*.

maxAbsCoeff(*(Vector6)arg1*) → float :

Maximum absolute value over all elements.

maxCoeff(*(Vector6)arg1*) → float :

Maximum value over all elements.

mean(*(Vector6)arg1*) → float :

Mean value over all elements.

minCoeff(*(Vector6)arg1*) → float :

Minimum value over all elements.

norm(*(Vector6)arg1*) → float :

Euclidean norm.

normalize(*(Vector6)arg1*) → None :

Normalize this object in-place.

normalized(*(Vector6)arg1*) → Vector6 :

Return normalized copy of this object

outer(*(Vector6)arg1*, *(Vector6)other*) → Matrix6 :

Outer product with *other*.

prod(*(Vector6)arg1*) → float :

Product of all elements.

pruned(*(Vector6)arg1*[, *(float)absTol=1e-06*]) → Vector6 :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

rows(*(Vector6)arg1*) → int :

Number of rows.

squaredNorm(*(Vector6)arg1*) → float :

Square of the Euclidean norm.

sum(*(Vector6)arg1*) → float :

Sum of all elements.

tail(*(Vector6)arg1*) → *Vector3*

class yade._minieigenHP.**Vector6c**

/TODO/

Identity = Vector6c(1,0,0, 0,0,0)

Ones = Vector6c(1,1,1, 1,1,1)

static Random() → Vector6c :

Return an object where all elements are randomly set to values between 0 and 1.

static Unit(*(int)arg1***)** → *Vector6c*

Zero = Vector6c(0,0,0, 0,0,0)

__init__(*(object)arg1***)** → None

___init___((object)arg1, (Vector6c)other) -> None

___init___((object)arg1, (complex)v0, (complex)v1, (complex)v2, (complex)v3, (complex)v4, (complex)v5) -> object

___init___((object)arg1, (Vector3c)head, (Vector3c)tail) -> object

asDiagonal(*(Vector6c)arg1***)** → Matrix6c :

Return diagonal matrix with this vector on the diagonal.

cols(*(Vector6c)arg1***)** → int :

Number of columns.

dot(*(Vector6c)arg1*, *(Vector6c)other***)** → complex :

Dot product with *other*.

head(*(Vector6c)arg1***)** → *Vector3c*

isApprox(*(Vector6c)arg1*, *(Vector6c)other*[, *(float)prec=1e-12*]**)** → bool :

Approximate comparison with precision *prec*.

maxAbsCoeff(*(Vector6c)arg1***)** → float :

Maximum absolute value over all elements.

mean(*(Vector6c)arg1***)** → complex :

Mean value over all elements.

norm(*(Vector6c)arg1***)** → float :

Euclidean norm.

normalize(*(Vector6c)arg1***)** → None :

Normalize this object in-place.

normalized(*(Vector6c)arg1***)** → Vector6c :

Return normalized copy of this object

outer(*(Vector6c)arg1*, *(Vector6c)other***)** → Matrix6c :

Outer product with *other*.

prod(*(Vector6c)arg1***)** → complex :

Product of all elements.

pruned(*(Vector6c)arg1*[, *(float)absTol=1e-06*]**)** → Vector6c :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

rows(*(Vector6c)arg1***)** → int :

Number of rows.

squaredNorm(*(Vector6c)arg1*) → float :
Square of the Euclidean norm.

sum(*(Vector6c)arg1*) → complex :
Sum of all elements.

tail(*(Vector6c)arg1*) → *Vector3c*

class yade._minieigenHP.Vector6i
6-dimensional float vector.

Supported operations (`f` if a float/int, `v` is a Vector6): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`.

Implicit conversion from sequence (list, tuple, …) of 6 ints.

Static attributes: `Zero`, `Ones`.

Identity = Vector6i(1,0,0, 0,0,0)

Ones = Vector6i(1,1,1, 1,1,1)

static Random() → Vector6i :
Return an object where all elements are randomly set to values between 0 and 1.

static Unit(*(int)arg1*) → *Vector6i*

Zero = Vector6i(0,0,0, 0,0,0)

__init__(*(object)arg1*) → None
___init___((object)arg1, (Vector6i)other) -> None
___init___((object)arg1, (int)v0, (int)v1, (int)v2, (int)v3, (int)v4, (int)v5) -> object
___init___((object)arg1, (Vector3i)head, (Vector3i)tail) -> object

asDiagonal(*(Vector6i)arg1*) → object :
Return diagonal matrix with this vector on the diagonal.

cols(*(Vector6i)arg1*) → int :
Number of columns.

dot(*(Vector6i)arg1*, *(Vector6i)other*) → int :
Dot product with *other*.

head(*(Vector6i)arg1*) → *Vector3i*

isApprox(*(Vector6i)arg1*, *(Vector6i)other*[, *(int)prec=0*]) → bool :
Approximate comparison with precision *prec*.

maxAbsCoeff(*(Vector6i)arg1*) → int :
Maximum absolute value over all elements.

maxCoeff(*(Vector6i)arg1*) → int :
Maximum value over all elements.

mean(*(Vector6i)arg1*) → int :
Mean value over all elements.

minCoeff(*(Vector6i)arg1*) → int :
Minimum value over all elements.

outer(*(Vector6i)arg1*, *(Vector6i)other*) → object :
Outer product with *other*.

prod(*(Vector6i)arg1*) → int :
Product of all elements.

rows(*(Vector6i)arg1*) → int :
Number of rows.

sum(*(Vector6i)arg1*) → int :
Sum of all elements.

tail(*(Vector6i)arg1*) → *Vector3i*

class yade._minieigenHP.**VectorX**
Dynamic-sized float vector.

Supported operations (`f` if a float/int, `v` is a VectorX): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`.

Implicit conversion from sequence (list, tuple, ...) of X floats.

static Ones(*(int)arg1*) → *VectorX*

static Random(*(int)len*) → VectorX :
Return vector of given length with all elements set to values between 0 and 1 randomly.

static Unit(*(int)arg1*, *(int)arg2*) → *VectorX*

static Zero(*(int)arg1*) → *VectorX*

__init__(*(object)arg1*) → None
___init___((object)arg1, (VectorX)other) -> None
___init___((object)arg1, (object)vv) -> object

asDiagonal(*(VectorX)arg1*) → MatrixX :
Return diagonal matrix with this vector on the diagonal.

cols(*(VectorX)arg1*) → int :
Number of columns.

dot(*(VectorX)arg1*, *(VectorX)other*) → float :
Dot product with *other*.

isApprox(*(VectorX)arg1*, *(VectorX)other*[, *(float)prec=1e-12*]) → bool :
Approximate comparison with precision *prec*.

maxAbsCoeff(*(VectorX)arg1*) → float :
Maximum absolute value over all elements.

maxCoeff(*(VectorX)arg1*) → float :
Maximum value over all elements.

mean(*(VectorX)arg1*) → float :
Mean value over all elements.

minCoeff(*(VectorX)arg1*) → float :
Minimum value over all elements.

norm(*(VectorX)arg1*) → float :
Euclidean norm.

normalize(*(VectorX)arg1*) → None :
Normalize this object in-place.

normalized(*(VectorX)arg1*) → VectorX :
Return normalized copy of this object

outer(*(VectorX)arg1*, *(VectorX)other*) → MatrixX :
Outer product with *other*.

prod(*(VectorX)arg1*) → float :
Product of all elements.

pruned(*(VectorX)arg1*[, *(float)absTol=1e-06*]) → VectorX :
Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

resize(*(VectorX)arg1*, *(int)arg2*) → None

rows(*(VectorX)arg1*) → int :
Number of rows.

squaredNorm(*(VectorX)arg1*) → float :
Square of the Euclidean norm.

sum(*(VectorX)arg1*) → float :
Sum of all elements.

class yade._minieigenHP.**VectorXc**
/TODO/

static Ones(*(int)arg1*) → *VectorXc*

static Random(*(int)len*) → VectorXc :
Return vector of given length with all elements set to values between 0 and 1 randomly.

static Unit(*(int)arg1*, *(int)arg2*) → *VectorXc*

static Zero(*(int)arg1*) → *VectorXc*

__init__(*(object)arg1*) → None
___init___((object)arg1, (VectorXc)other) -> None
___init___((object)arg1, (object)vv) -> object

asDiagonal(*(VectorXc)arg1*) → MatrixXc :
Return diagonal matrix with this vector on the diagonal.

cols(*(VectorXc)arg1*) → int :
Number of columns.

dot(*(VectorXc)arg1*, *(VectorXc)other*) → complex :
Dot product with *other*.

isApprox(*(VectorXc)arg1*, *(VectorXc)other*[, *(float)prec=1e-12*]) → bool :
Approximate comparison with precision *prec*.

maxAbsCoeff(*(VectorXc)arg1*) → float :
Maximum absolute value over all elements.

mean(*(VectorXc)arg1*) → complex :
Mean value over all elements.

norm(*(VectorXc)arg1*) → float :
Euclidean norm.

normalize(*(VectorXc)arg1*) → None :
Normalize this object in-place.

normalized(*(VectorXc)arg1*) → VectorXc :

Return normalized copy of this object

outer(*(VectorXc)arg1*, *(VectorXc)other*) → MatrixXc :

Outer product with *other*.

prod(*(VectorXc)arg1*) → complex :

Product of all elements.

pruned(*(VectorXc)arg1*[, *(float)absTol=1e-06*]) → VectorXc :

Zero all elements which are greater than *absTol*. Negative zeros are not pruned.

resize(*(VectorXc)arg1*, *(int)arg2*) → None

rows(*(VectorXc)arg1*) → int :

Number of rows.

squaredNorm(*(VectorXc)arg1*) → float :

Square of the Euclidean norm.

sum(*(VectorXc)arg1*) → complex :

Sum of all elements.

2.4.11 yade.mpy module

This module defines mpirun(), a parallel implementation of run() using a distributed memory approach. Message passing is done with mpi4py mainly, however some messages are also handled in c++ (with openmpi).

Note

Many internals of the mpy module listed on this page are not helpful to the user. Instead, please find *introductory material on mpy module* in user manual.

Logic:

The logic for an initially centralized scene is as follows:

1. Instanciate a complete, ordinary, yade scene
2. Insert subdomains as special yade bodies. This is somehow similar to adding a clump body on the top of clump members
3. Broadcast this scene to all workers. In the initialization phase the workers will:
   - define the bounding box of their assigned bodies and return it to other workers
   - detect which assigned bodies are virtually in interaction with other domains (based on their bounding boxes) and communicate the lists to the relevant workers
   - erase the bodies which are neither assigned nor virtually interacting with the subdomain
4. Run a number of 'regular' iterations without re-running collision detection (verlet dist mechanism). In each regular iteration the workers will:
   - calculate internal and cross-domains interactions
   - execute Newton on assigned bodies (modified Newton skips other domains)
   - send updated positions to other workers and partial force on floor to master
5. When one worker triggers collision detection all workers will follow. It will result in updating the intersections between subdomains.

6. If enabled, bodies may be re-allocated to different domains just after a collision detection, based on a filter. Custom filters are possible. One is predidefined here (medianFilter)

Rules:

- intersections[0] has 0-bodies (to which we need to send force)
- intersections[thisDomain] has ids of the other domains overlapping the current ones
- intersections[otherDomain] has ids of bodies in _current_ domain which are overlapping with other domain (for which we need to send updated pos/vel)

Hint:

handle subD.intersections with care (same for mirrorIntersections). subD.intersections.append() will not reach the c++ object. subD.intersections can only be assigned (a list of list of int)

yade.mpy.**MAX_RANK_OUTPUT = 5**

larger ranks will be skipped in mprint

yade.mpy.**REALLOCATE_FILTER**(*i*, *j*, *giveAway*)

Returns bodies in "i" to be assigned to "j" based on median split between the center points of subdomain's AABBs If giveAway!=0, positive or negative, "i" will give/acquire this number to "j" with nothing in return (for load balancing purposes)

class yade.mpy.**Timing_comm**(*inherits object*)

Allgather(*timing_name*, **args*, ***kwargs*)

Gather(*timing_name*, **args*, ***kwargs*)

Gatherv(*timing_name*, **args*, ***kwargs*)

allreduce(*timing_name*, **args*, ***kwargs*)

bcast(*timing_name*, **args*, ***kwargs*)

clear()

enable_timing()

mpiSendStates(*timing_name*, **args*, ***kwargs*)

mpiWait(*timing_name*, **args*, ***kwargs*)

mpiWaitReceived(*timing_name*, **args*, ***kwargs*)

print_all()

recv(*timing_name*, **args*, ***kwargs*)

send(*timing_name*, **args*, ***kwargs*)

yade.mpy.**bodyErase**(*ids*)

The parallel version of O.bodies.erase(id), should be called collectively else the distributed scenes become inconsistent with each other (even the subdomains which don't have 'id' can call safely). For performance, better call on a list: bodyErase([i,j,k]).

yade.mpy.**checkAndCollide**()

return true if collision detection needs activation in at least one SD, else false. If COPY_MIRROR_BODIES_WHEN_COLLIDE run collider when needed, and in that case return False.

yade.mpy.**colorDomains**()

Apply color to body to reflect their subdomain idx

yade.mpy.**configure**()

Import MPI and define context, configure will no spawn workers by itself, that is done by initialize() openmpi environment variables needs to be set before calling configure()

yade.mpy.**declareMasterInteractive**()

This is to signal that we are in interactive session, so TIMEOUT will be reset to 0 (ignored)

yade.mpy.**disconnect**()

Kill all mpi processes, leaving python interpreter to rank 0 as in single-threaded execution. The scenes in workers are lost since further reconnexion to mpi will just spawn new processes. The scene in master thread is left unchanged.

yade.mpy.**eraseRemote**()

yade.mpy.**genLocalIntersections**(*subdomains*)

Defines sets of bodies within current domain overlapping with other domains.

The structure of the data for domain 'k' is: [[id1, id2, ...], <———— intersections[0] = ids of bodies in domain k interacting with master domain (subdomain k itself excluded) [id3, id4, ...], <———— intersections[1] = ids of bodies in domain k interacting with domain rank=1 (subdomain k itself excluded) ... [domain1, domain2, domain3, ...], <————- intersections[k] = ranks (not ids!) of external domains interacting with domain k ...]

yade.mpy.**genUpdatedStates**(*b_ids*)

return list of [id,state] (or [id,state,shape] conditionnaly) to be sent to other workers

yade.mpy.**initialize**(*np*)

yade.mpy.**isendRecvForces**()

Communicate forces from subdomain to master Warning: the sending sides (everyone but master) must wait() the returned list of requests

yade.mpy.**makeColorScale**(*n=None*)

yade.mpy.**makeMpiArgv**()

yade.mpy.**maskedConnection**(*b*, *boolArray*)

List bodies within a facet selectively, the ones marked 'True' in boolArray (i.e. already selected from another facet) are discarded

yade.mpy.**maskedPFacet**(*b*, *boolArray*)

List bodies within a facet selectively, the ones marked 'True' in boolArray (i.e. already selected from another facet) are discarded

yade.mpy.**medianFilter**(*i*, *j*, *giveAway*)

Returns bodies in "i" to be assigned to "j" based on median split between the center points of subdomain's AABBs If giveAway!=0, positive or negative, "i" will give/acquire this number to "j" with nothing in return (for load balancing purposes)

yade.mpy.**mergeScene**()

yade.mpy.**migrateBodies**(*ids*, *origin*, *destination*)

Reassign bodies from origin to destination. The function has to be called by both origin (send) and destination (recv). Note: subD.completeSendBodies() will have to be called after a series of reassignement since subD.sendBodies() is non-blocking

yade.mpy.**mpiStats**()

yade.mpy.**mpirun**(*nSteps*, *np=None*, *withMerge=False*)

Parallel version of O.run() using MPI domain decomposition.

Parameters

nSteps : The numer of steps to compute np : number of mpi workers (master+subdomains), if=1 the function fallback to O.run() withMerge : wether subdomains should be merged into master at the end of the run (default False). If True the scene in the master process is exactly in the same state as after O.run(nSteps,True). The merge can be time consumming, it is recommended to activate only if post-processing or other similar tasks require it.

yade.mpy.**mprint**(**args*, *force=False*)

Print with rank-reflecting color regardless of mpy.VERBOSE_OUTPUT, still limited to rank<=mpy.MAX_RANK_OUTPUT

yade.mpy.**pairOp**(*talkTo*)

yade.mpy.**parallelCollide**()

yade.mpy.**probeRecvMessage**(*source*, *tag*)

yade.mpy.**projectedBounds**(*i*, *j*)

Returns sorted list of projections of bounds on a given axis, with bounds taken in i->j and j->i intersections

yade.mpy.**reallocateBodiesPairWiseBlocking**(*_filter*, *otherDomain*)

Re-assign bodies from/to otherDomain based on '_filter' argument. Requirement: '_filter' is a function taking ranks of origin and destination and returning the list of bodies (by index) to be moved. That's where the decomposition strategy is defined. See example medianFilter (used by default).

yade.mpy.**reallocateBodiesToSubdomains**(*_filter=<function medianFilter>*, *blocking=True*)

Re-assign bodies to subdomains based on '_filter' argument. Requirement: '_filter' is a function taking ranks of origin and destination and returning the list of bodies (by index) to be moved. That's where the decomposition strategy is defined. See example medianFilter (used by default). This function must be called in parallel, hence if ran interactively the command needs to be sent explicitely: mp.sendCommand("all","reallocateBodiesToSubdomains(medianFilter)",True)

yade.mpy.**reboundRemoteBodies**(*ids*)

update states of bodies handled by other workers, argument 'states' is a list of [id,state] (or [id,state,shape] conditionnaly)

yade.mpy.**receiveForces**(*subdomains*)

Accumulate forces from subdomains (only executed by master process), should happen after ForceResetter but before Newton and before any other force-dependent engine (e.g. StressController), could be inserted via yade's pyRunner.

yade.mpy.**recordMpiTiming**(*name*, *val*)

append val to a list of values defined by 'name' in the dictionnary timing.mpi

yade.mpy.**runOnSynchronouslPairs**(*workers*, *command*)

Locally (from one worker POV), this function runs interactive mpi tasks defined by 'command' on a list of other workers (typically the list of interacting subdomains). Overall, peer-to-peer connexions are established so so that 'command' is executed symmetrically and simultaneously on both sides of each worker pair. I.e. if worker "i" executes "command" with argument "j" (index of another worker), then by design "j" will execute the same thing with argument "i" *simultaneously.*

In many cases a similar series of data exchanges can be obtained more simply (and fastly) with asynchronous irecv+send like below.

for w in workers:
m=comm.irecv(w) comm.send(data,dest=w)

The above only works if the messages are all known in advance locally, before any communication. If the interaction with workers[1] depends on the result of a previous interaction with workers[0] OTOH, it needs synchronous execution, hence this function. Synchronicity is also required if more than one blocking call is present in 'command', else an obvious deadlock as if 'irecv' was replaced

by 'recv' in that naive loop. Both cases occur with the 'medianFilter' algorithm, hence why we need this synchronous method.

In this function pair connexions are established by the workers in a non-supervized and non-deterministic manner. Each time an interactive communication (i,j) is established 'command' is executed simultaneously by i and j. It is guaranted that all possible pairs are visited.

The function can be used for all-to-all operations (N^2 pairs), but more interestingly it works with workers=intersections[rank] (O(N) pairs). It can be tested with the dummy funtion 'pairOp': runOnSynchronouslPairs(range(numThreads),pairOp)

command:
a function taking index of another worker as argument, can include blocking communications with the other worker since runOnSynchronouslPairs guarantee that the other worker will be running the command symmetrically.

yade.mpy.**sendCommand**(*executors*, *command*, *wait=True*, *workerToWorker=False*)

Send a command to a worker (or list of) from master or from another worker. Accepted executors are "i", "[i,j,k]", "slaves", "all" (then even master will execute the command).

yade.mpy.**sendRecvStates**()

yade.mpy.**shrinkIntersections**()

Reduce intersections and mirrorIntersections to bodies effectively interacting with another statefull body form current subdomain This will reduce the number of updates in sendRecvStates Initial lists are backed-up and need to be restored (and all states updated) before collision detection (see checkAndCollide())

yade.mpy.**spawnedProcessWaitCommand**()

yade.mpy.**splitScene**()

Split a monolithic scene into distributed scenes on threads.

Precondition: the bodies have subdomain no. set in user script

yade.mpy.**unboundRemoteBodies**()

Turn bounding boxes on/off depending on rank

yade.mpy.**updateAllIntersections**()

yade.mpy.**updateDomainBounds**(*subdomains*)

Update bounds of current subdomain, broadcast, and receive updated bounds from other subdomains Precondition: collider.boundDispatcher.___call___()

yade.mpy.**updateMirrorOwners**()

yade.mpy.**updateRemoteStates**(*states*, *setBounded=False*)

update states of bodies handled by other workers, argument 'states' is a list of [id,state] (or [id,state,shape] conditionnaly)

yade.mpy.**waitForces**()

wait until all forces are sent to master. O.freqs is empty for master, and for all threads if not ACCUMULATE_FORCES

yade.mpy.**wprint**(**args*)

Print with rank-reflecting color, *only if* mpy.VERBOSE_OUTPUT=True (else see *mpy.mprint*), limited to rank<=mpy.MAX_RANK_OUTPUT

2.4.12 yade.pack module

Creating packings and filling volumes defined by boundary representation or constructive solid geometry.

For examples, see

- examples/gts-horse/gts-operators.py
- examples/gts-horse/gts-random-pack-obb.py
- examples/gts-horse/gts-random-pack.py
- examples/test/pack-cloud.py
- examples/test/pack-predicates.py
- examples/packs/packs.py
- examples/gts-horse/gts-horse.py
- examples/WireMatPM/wirepackings.py

yade.pack.**SpherePack_toSimulation**(*self*, *rot=Matrix3(1, 0, 0, 0, 1, 0, 0, 0, 1)*, ***kw*)

Append spheres directly to the simulation. In addition calling *O.bodies.append*, this method also appropriately sets periodic cell information of the simulation.

```
>>> from yade import pack; from math import *
>>> sp=pack.SpherePack()
```

Create random periodic packing with 20 spheres:

```
>>> sp.makeCloud((0,0,0),(5,5,5),rMean=.5,rRelFuzz=.5,periodic=True,num=20)
20
```

Virgin simulation is aperiodic:

```
>>> O.reset()
>>> O.periodic
False
```

Add generated packing to the simulation, rotated by 45° along +z

```
>>> sp.toSimulation(rot=Quaternion((0,0,1),pi/4),color=(0,0,1))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Periodic properties are transferred to the simulation correctly, including rotation (this could be avoided by explicitly passing "hSize=O.cell.hSize" as an argument):

```
>>> O.periodic
True
>>> O.cell.refSize
Vector3(5,5,5)
>>> O.cell.hSize
Matrix3(3.53553,-3.53553,0, 3.53553,3.53553,0, 0,0,5)
```

The current state (even if rotated) is taken as mechanically undeformed, i.e. with identity transformation:

```
>>> O.cell.trsf
Matrix3(1,0,0, 0,1,0, 0,0,1)
```

Parameters

- **rot** (*Quaternion/Matrix3*) – rotation of the packing, which will be applied on spheres and will be used to set *Cell.trsf* as well.
- ****kw** – passed to *utils.sphere*

Returns

list of body ids added (like *O.bodies.append*)

yade.pack.**filterSpherePack**(*predicate*, *spherePack*, *returnSpherePack=None*, ***kw*)

Using given SpherePack instance, return spheres that satisfy predicate. It returns either a *pack.SpherePack* (if returnSpherePack) or a list. The packing will be recentered to match the predicate and warning is given if the predicate is larger than the packing.

yade.pack.**gtsSurface2Facets**(*surf*, ***kw*)

Construct facets from given GTS surface. **kw is passed to utils.facet.

yade.pack.**gtsSurfaceBestFitOBB**(*surf*)

Return (Vector3 center, Vector3 halfSize, Quaternion orientation) describing best-fit oriented bounding box (OBB) for the given surface. See cloudBestFitOBB for details.

yade.pack.**hexaNet**(*radius*, *cornerCoord=[0, 0, 0]*, *xLength=1.0*, *yLength=0.5*, *mos=0.08*, *a=0.04*, *b=0.04*, *startAtCorner=True*, *isSymmetric=False*, ***kw*)

Definition of the particles for a hexagonal wire net in the x-y-plane for the WireMatPM.

Parameters

- **radius** – radius of the particle
- **cornerCoord** – coordinates of the lower left corner of the net
- **xLenght** – net length in x-direction
- **yLenght** – net length in y-direction
- **mos** – mesh opening size (horizontal distance between the double twists)
- **a** – length of double-twist
- **b** – height of single wire section
- **startAtCorner** – if true the generation starts with a double-twist at the lower left corner
- **isSymmetric** – defines if the net is symmetric with respect to the y-axis

Returns

set of spheres which defines the net (net) and exact dimensions of the net (lx,ly).

Note

This packing works for the WireMatPM only. The particles at the corner are always generated first. For examples on how to use this packing see examples/WireMatPM. In order to create the proper interactions for the net the interaction radius has to be adapted in the simulation.

class yade.pack.**inConvexPolyhedron**(*inherits Predicate*)

aabb(*(Predicate)arg1*) → tuple :

lower and upper corner of predicate's axis aligned bounding box

aabb((Predicate)arg1) -> None

center(*(Predicate)arg1*) → yade._minieigenHP.Vector3 :

center of the predicate

containsPoint(*(Predicate)arg1*, *(yade._minieigenHP.Vector3)pt*[, *(float)pad=0*]) → bool :

if given point is inside the predicate or not. `pred.containsPoint(pt,pad)` is equivalent to directly calling predicate itself `pred(pt,pad)`

containsPoint((Predicate)arg1, (yade._minieigenHP.Vector3)pt [, (float)pad=0]) -> None

dim(*(Predicate)arg1*) → yade._minieigenHP.Vector3 :

axis aligned dimensions of the predicate

class yade.pack.**inGtsSurface_py**(*inherits Predicate*)

This class was re-implemented in c++, but should stay here to serve as reference for implementing Predicates in pure python code. C++ allows us to play dirty tricks in GTS which are not accessible through pygts itself; the performance penalty of pygts comes from fact that if constructs and destructs bb tree for the surface at every invocation of gts.Point().is_inside(). That is cached in the c++ code, provided that the surface is not manipulated with during lifetime of the object (user's responsibility).

—

Predicate for GTS surfaces. Constructed using an already existing surfaces, which must be closed.

import gts surf=gts.read(open('horse.gts')) inGtsSurface(surf)

Note

Padding is optionally supported by testing 6 points along the axes in the pad distance. This must be enabled in the ctor by saying doSlowPad=True. If it is not enabled and pad is not zero, warning is issued.

aabb(*(Predicate)arg1*) → tuple :

lower and upper corner of predicate's axis aligned bounding box

aabb((Predicate)arg1) -> None

center(*(Predicate)arg1*) → yade._minieigenHP.Vector3 :

center of the predicate

containsPoint(*(Predicate)arg1*, *(yade._minieigenHP.Vector3)pt*[, *(float)pad=0*]) → bool :

if given point is inside the predicate or not. `pred.containsPoint(pt,pad)` is equivalent to directly calling predicate itself `pred(pt,pad)`

containsPoint((Predicate)arg1, (yade._minieigenHP.Vector3)pt [, (float)pad=0]) -> None

dim(*(Predicate)arg1*) → yade._minieigenHP.Vector3 :

axis aligned dimensions of the predicate

class yade.pack.**inHalfSpace**(*inherits Predicate*)

Predicate returning True any points, with infinite bounding box.

aabb(*(Predicate)arg1*) → tuple :

lower and upper corner of predicate's axis aligned bounding box

aabb((Predicate)arg1) -> None

center(*(Predicate)arg1*) → yade._minieigenHP.Vector3 :

center of the predicate

containsPoint(*(Predicate)arg1*, *(yade._minieigenHP.Vector3)pt*[, *(float)pad=0*]) → bool :

if given point is inside the predicate or not. `pred.containsPoint(pt,pad)` is equivalent to directly calling predicate itself `pred(pt,pad)`

containsPoint((Predicate)arg1, (yade._minieigenHP.Vector3)pt [, (float)pad=0]) -> None

dim(*(Predicate)arg1*) → yade._minieigenHP.Vector3 :

axis aligned dimensions of the predicate

class yade.pack.**inSpace**(*inherits Predicate*)

Predicate returning True for any points, with infinite bounding box.

`aabb`(*(Predicate)arg1*) → tuple :

lower and upper corner of predicate's axis aligned bounding box

aabb((Predicate)arg1) -> None

`center`(*(Predicate)arg1*) → yade._minieigenHP.Vector3 :

center of the predicate

`containsPoint`(*(Predicate)arg1*, *(yade._minieigenHP.Vector3)pt*[, *(float)pad=0*]) → bool :

if given point is inside the predicate or not. `pred.containsPoint(pt,pad)` is equivalent to directly calling predicate itself `pred(pt,pad)`

containsPoint((Predicate)arg1, (yade._minieigenHP.Vector3)pt [, (float)pad=0]) -> None

`dim`(*(Predicate)arg1*) → yade._minieigenHP.Vector3 :

axis aligned dimensions of the predicate

yade.pack.`randomDensePack`(*predicate*, *radius*, *material=-1*, *dim=None*, *cropLayers=0*, *rRelFuzz=0.0*, *spheresInCell=0*, *memoizeDb=None*, *useOBB=False*, *memoDbg=False*, *color=None*, *returnSpherePack=None*, *seed=-1*)

Generator of random dense packing with given geometry properties, using TriaxialTest (aperiodic) or PeriIsoCompressor (periodic). The periodicity depens on whether the spheresInCell parameter is given.

O.switchScene() magic is used to have clean simulation for TriaxialTest without deleting the original simulation. This function therefore should never run in parallel with some code accessing your simulation.

Parameters

- `predicate` – solid-defining predicate for which we generate packing
- `spheresInCell` – if given, the packing will be periodic, with given number of spheres in the periodic cell.
- `radius` – mean radius of spheres
- `rRelFuzz` – relative fuzz of the radius – e.g. radius=10, rRelFuzz=.2, then spheres will have radii 10 ± (10*.2)), with an uniform distribution. 0 by default, meaning all spheres will have exactly the same radius.
- `cropLayers` – (aperiodic only) how many layers of spheres will be added to the computed dimension of the box so that there no (or not so much, at least) boundary effects at the boundaries of the predicate.
- `dim` – dimension of the packing, to override dimensions of the predicate (if it is infinite, for instance)
- `memoizeDb` – name of sqlite database (existent or nonexistent) to find an already generated packing or to store the packing that will be generated, if not found (the technique of caching results of expensive computations is known as memoization). Fuzzy matching is used to select suitable candidate – packing will be scaled, rRelFuzz and dimensions compared. Packing that are too small are dictarded. From the remaining candidate, the one with the least number spheres will be loaded and returned.
- `useOBB` – effective only if a inGtsSurface predicate is given. If true (not default), oriented bounding box will be computed first; it can reduce substantially number of spheres for the triaxial compression (like 10× depending on how much asymmetric the body is), see examples/gts-horse/gts-random-pack-obb.py
- `memoDbg` – show packings that are considered and reasons why they are rejected/accepted
- `returnSpherePack` – see the corresponding argument in *pack.filterSpherePack*

Returns
SpherePack object with spheres, filtered by the predicate.

yade.pack.**randomPeriPack**(*radius*, *initSize*, *rRelFuzz=0.0*, *memoizeDb=None*, *noPrint=False*, *seed=-1*)

Generate periodic dense packing.

A cell of initSize is stuffed with as many spheres as possible, then we run periodic compression with PeriIsoCompressor, just like with randomDensePack.

Parameters

- **radius** – mean sphere radius
- **rRelFuzz** – relative fuzz of sphere radius (equal distribution); see the same param for randomDensePack.
- **initSize** – initial size of the periodic cell.

Returns
SpherePack object, which also contains periodicity information.

yade.pack.**regularHexa**(*predicate*, *radius*, *gap*, ***kw*)

Return set of spheres in regular hexagonal grid, clipped inside solid given by predicate. Created spheres will have given radius and will be separated by gap space.

yade.pack.**regularOrtho**(*predicate*, *radius*, *gap*, ***kw*)

Return set of spheres in regular orthogonal grid, clipped inside solid given by predicate. Created spheres will have given radius and will be separated by gap space.

yade.pack.**revolutionSurfaceMeridians**(*sects*, *angles*, *origin=Vector3(0, 0, 0)*, *orientation=Quaternion((1, 0, 0), 0)*)

Revolution surface given sequences of 2d points and sequence of corresponding angles, returning sequences of 3d points representing meridian sections of the revolution surface. The 2d sections are turned around z-axis, but they can be transformed using the origin and orientation arguments to give arbitrary orientation.

yade.pack.**sweptPolylines2gtsSurface**(*pts*, *threshold=0*, *capStart=False*, *capEnd=False*)

Create swept suface (as GTS triangulation) given same-length sequences of points (as 3-tuples).

If threshold is given (>0), then

- degenerate faces (with edges shorter than threshold) will not be created
- gts.Surface().cleanup(threshold) will be called before returning, which merges vertices mutually closer than threshold. In case your pts are closed (last point concident with the first one) this will the surface strip of triangles. If you additionally have capStart==True and capEnd==True, the surface will be closed.

Note

capStart and capEnd make the most naive polygon triangulation (diagonals) and will perhaps fail for non-convex sections.

Warning

the algorithm connects points sequentially; if two polylines are mutually rotated or have inverse sense, the algorithm will not detect it and connect them regardless in their given order.

Creation, manipulation, IO for generic sphere packings.

class yade._packSpheres.SpherePack

Set of spheres represented as centers and radii. This class is returned by *pack.randomDensePack*, *pack.randomPeriPack* and others. The object supports iteration over spheres, as in

```
>>> sp=SpherePack()
>>> for center,radius in sp: print(center,radius)
```

```
>>> for sphere in sp: print(sphere[0],sphere[1])   ## same, but without␣
↪unpacking the tuple automatically
```

```
>>> for i in range(0,len(sp)): print(sp[i][0], sp[i][1])   ## same, but␣
↪accessing spheres by index
```

Special constructors

Construct from list of `[(c1,r1),(c2,r2),…]`. To convert two same-length lists of `centers` and `radii`, construct with `zip(centers,radii)`.

`__init__`(*(object)arg1*[, *(list)list*]) → None :

Empty constructor, optionally taking list [((cx,cy,cz),r), …] for initial data.

`aabb`(*(SpherePack)arg1*) → tuple :

Get axis-aligned bounding box coordinates, as 2 3-tuples.

`add`(*(SpherePack)arg1*, *(yade._minieigenHP.Vector3)arg2*, *(float)arg3*) → None :

Add single sphere to packing, given center as 3-tuple and radius

`property appliedPsdScaling`

A factor between 0 and 1, uniformly applied on all sizes of of the PSD.

`cellFill`(*(SpherePack)arg1*, *(yade._minieigenHP.Vector3)arg2*) → None :

Repeat the packing (if periodic) so that the results has dim() >= given size. The packing retains periodicity, but changes cellSize. Raises exception for non-periodic packing.

`cellRepeat`(*(SpherePack)arg1*, *(yade._minieigenHP.Vector3i)arg2*) → None :

Repeat the packing given number of times in each dimension. Periodicity is retained, cellSize changes. Raises exception for non-periodic packing.

`property cellSize`

Size of periodic cell; is Vector3(0,0,0) if not periodic. (Change this property only if you know what you're doing).

`center`(*(SpherePack)arg1*) → yade._minieigenHP.Vector3 :

Return coordinates of the bounding box center.

`dim`(*(SpherePack)arg1*) → yade._minieigenHP.Vector3 :

Return dimensions of the packing in terms of aabb(), as a 3-tuple.

`fromList`(*(SpherePack)arg1*, *(list)arg2*) → None :

Make packing from given list, same format as for constructor. Discards current data.

fromList((SpherePack)arg1, (object)centers, (object)radii) -> None :

Make packing from given list, same format as for constructor. Discards current data.

`fromSimulation`(*(SpherePack)arg1*) → None :

Make packing corresponding to the current simulation. Discards current data.

getClumps(*(SpherePack)arg1*) → tuple :

Return lists of sphere ids sorted by clumps they belong to. The return value is (standalones,[clump1,clump2,…]), where each item is list of id's of spheres.

hasClumps(*(SpherePack)arg1*) → bool :

Whether this object contains clumps.

property isPeriodic

was the packing generated in periodic boundaries?

load(*(SpherePack)arg1*, *(str)fileName*) → None :

Load packing from external text file (current data will be discarded).

makeCloud(*(SpherePack)arg1*[, *(yade._minieigenHP.Vector3)minCorner=Vector3(0, 0, 0)*[, *(yade._minieigenHP.Vector3)maxCorner=Vector3(0, 0, 0)*[, *(float)rMean=-1*[, *(float)rRelFuzz=0*[, *(int)num=-1*[, *(bool)periodic=False*[, *(float)porosity=0.65*[, *(object)psdSizes=[]*[, *(object)psdCumm=[]*[, *(bool)distributeMass=False*[, *(int)seed=-1*[, *(yade._minieigenHP.Matrix3)hSize=Matrix3(0, 0, 0, 0, 0, 0, 0, 0, 0)*]]]]]]]]]]]]) → int :

Create a random cloud of particles enclosed in a parallelepiped. The resulting packing is a gas-like state with no contacts between particles initially. Usually used as a first step before reaching a dense packing.

Parameters

- **minCorner** (Vector3) – lower corner of an axis-aligned box
- **maxCorner** (Vector3) – upper corner of an axis-aligned box
- **hSize** (Matrix3) – base vectors of a generalized box (arbitrary parallelepiped, typically *Cell::hSize*), superseeds minCorner and maxCorner if defined. For periodic boundaries only.
- **rMean** (*float*) – mean radius or spheres
- **rRelFuzz** (*float*) – dispersion of radius relative to rMean
- **num** (*int*) – number of spheres to be generated. If negative (default), generate as many as possible with stochastic sizes, ending after a fixed number of tries to place the sphere in space, else generate exactly **num** spheres with deterministic size distribution.
- **periodic** (*bool*) – whether the packing to be generated should be periodic
- **porosity** (*float*) – initial guess for the iterative generation procedure (if **num**>1). The algorithm will be retrying until the number of generated spheres is **num**. The first iteration tries with the provided porosity, but next iterations increase it if necessary (hence an initialy high porosity can speed-up the algorithm). If **psdSizes** is not defined, **rRelFuzz** (z) and **num** (N) are used so that the porosity given (ρ) is approximately achieved at the end of generation, $r_m = \sqrt[3]{\frac{V(1-\rho)}{\frac{4}{3}\pi(1+z^2)N}}$. The default is ρ=0.5. The optimal value depends on **rRelFuzz** or **psdSizes**.
- **psdSizes** – sieve sizes (particle diameters) when particle size distribution (PSD) is specified.
- **psdCumm** – cummulative fractions of particle sizes given by **psdSizes**; must be the same length as *psdSizes* and should be non-decreasing.
- **distributeMass** (*bool*) – if **True**, given distribution reflects mass per radius (the most common), else number of spheres per radius.
- **seed** – number used to initialize the random number generator.

Returns

number of created spheres, which can be lower than `num` depending on the method used.

Note

- Works in 2D if `minCorner[k]=maxCorner[k]` for one coordinate.
- If `num` is defined, then sizes generation is deterministic, giving the best fit of target distribution. It enables spheres placement in descending size order, thus giving lower porosity than the random generation.
- By default (with `distributeMass==False`), the distribution is applied to particle count (i.e. particle count percent passing). The typical geomechanics sense of "particle size distribution" is the distribution of *mass fraction* (i.e. mass percent passing); this can be achieved with `distributeMass=True`.
- Sphere radius distribution can be specified using one of the following ways:
  1. `rMean`, `rRelFuzz` and `num` gives uniform radius distribution in $\texttt{rMean}\times(1\pm\texttt{rRelFuzz})$. Less than `num` spheres can be generated if it is too high.
  2. `rRelFuzz`, `num` and (optional) `porosity`, which estimates mean radius so that `porosity` is attained at the end. `rMean` must be less than 0 (default). `porosity` is only an initial guess for the generation algorithm, which will retry with higher porosity until the prescibed *num* is obtained.
  3. `psdSizes` and `psdCumm`, two arrays specifying points of the particle size distribution function. As many spheres as possible are generated.
  4. `psdSizes`, `psdCumm`, `num`, and (optional) `porosity`, like above but if `num` is not obtained, `psdSizes` will be scaled down uniformly, until `num` is obtained (see *appliedPsdScaling*).

`makeClumpCloud`(*(SpherePack)arg1*, *(yade._minieigenHP.Vector3)minCorner*, *(yade._minieigenHP.Vector3)maxCorner*, *(object)clumps*[, *(bool)periodic=False*[, *(int)num=-1*[, *(int)seed=-1*]]]) → int :

Create a random (in particles positions and orientations) cloud of clumps the same way makeCloud does with spheres. The parameters `minCorner`, `maxCorner`, `periodic`, `num` and `seed` are the same as in makeCloud. The parameter `clumps` is a list containing all the different clumps to be appended as `SpherePack` objects. Here is an exemple that shows how to create a cloud made of 10 identical clumps :

```
clp = SpherePack([((0,0,0), 1e-2), ((1e-2,0,0), 1e-2)]) # The clump we want␣
↪a cloud of
sp = SpherePack()
sp.makeClumpCloud((0,0,0), (1,1,1), [clp], num=10, seed=42)
sp.toSimulation() # All the particles in the cloud are now appended to␣
↪O.bodies
```

`psd`(*(SpherePack)arg1*[, *(int)bins=50*[, *(bool)mass=True*]]) → tuple :

Return particle size distribution of the packing.

Parameters

- `bins` (*int*) – number of bins between minimum and maximum diameter
- `mass` – Compute relative mass rather than relative particle count for each bin. Corresponds to *distributeMass parameter for makeCloud*.

Returns

tuple of `(cumm,edges)`, where `cumm` are cummulative fractions for respective diameters and `edges` are those diameter values. Dimension of both arrays is equal to `bins+1`.

relDensity(*(SpherePack)arg1*) → float :

Relative packing density, measured as sum of spheres' volumes / aabb volume. (Sphere overlaps are ignored.)

rotate(*(SpherePack)arg1*, *(yade._minieigenHP.Vector3)axis*, *(float)angle*) → None :

Rotate all spheres around packing center (in terms of aabb()), given axis and angle of the rotation.

save(*(SpherePack)arg1*, *(str)fileName*) → None :

Save packing to external text file (will be overwritten) in a 'x y z r *clumpId*' format.

scale(*(SpherePack)arg1*, *(float)arg2*) → None :

Scale the packing around its center (in terms of aabb()) by given factor (may be negative).

toList(*(SpherePack)arg1*) → list :

Return packing data as python list.

toSimulation(*rot=Matrix3(1, 0, 0, 0, 1, 0, 0, 0, 1)*, ***kw*)

Append spheres directly to the simulation. In addition calling *O.bodies.append*, this method also appropriately sets periodic cell information of the simulation.

```
>>> from yade import pack; from math import *
>>> sp=pack.SpherePack()
```

Create random periodic packing with 20 spheres:

```
>>> sp.makeCloud((0,0,0),(5,5,5),rMean=.5,rRelFuzz=.5,periodic=True,num=20)
20
```

Virgin simulation is aperiodic:

```
>>> O.reset()
>>> O.periodic
False
```

Add generated packing to the simulation, rotated by 45° along +z

```
>>> sp.toSimulation(rot=Quaternion((0,0,1),pi/4),color=(0,0,1))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Periodic properties are transferred to the simulation correctly, including rotation (this could be avoided by explicitly passing "hSize=O.cell.hSize" as an argument):

```
>>> O.periodic
True
>>> O.cell.refSize
Vector3(5,5,5)
>>> O.cell.hSize
Matrix3(3.53553,-3.53553,0, 3.53553,3.53553,0, 0,0,5)
```

The current state (even if rotated) is taken as mechanically undeformed, i.e. with identity transformation:

```
>>> O.cell.trsf
Matrix3(1,0,0, 0,1,0, 0,0,1)
```

Parameters

- **rot** (*Quaternion/Matrix3*) – rotation of the packing, which will be applied on spheres and will be used to set *Cell.trsf* as well.
- ****kw** – passed to *utils.sphere*

Returns
list of body ids added (like *O.bodies.append*)

translate(*(SpherePack)arg1*, *(yade._minieigenHP.Vector3)arg2*) → None :
Translate all spheres by given vector.

class yade._packSpheres.**SpherePackIterator**

__init__(*(object)arg1*, *(SpherePackIterator)arg2*) → None

next()
___next___((SpherePackIterator)arg1) -> tuple

Spatial predicates for volumes (defined analytically or by triangulation).

class yade._packPredicates.**Predicate**
Spatial predicate base class. Predicates support boolean operations as described in user's manual

aabb(*(Predicate)arg1*) → tuple :
lower and upper corner of predicate's axis aligned bounding box

aabb((Predicate)arg1) -> None

center(*(Predicate)arg1*) → yade._minieigenHP.Vector3 :
center of the predicate

containsPoint(*(Predicate)arg1*, *(yade._minieigenHP.Vector3)pt*[, *(float)pad=0*]) → bool :
if given point is inside the predicate or not. `pred.containsPoint(pt,pad)` is equivalent to directly calling predicate itself `pred(pt,pad)`

containsPoint((Predicate)arg1, (yade._minieigenHP.Vector3)pt [, (float)pad=0]) -> None

dim(*(Predicate)arg1*) → yade._minieigenHP.Vector3 :
axis aligned dimensions of the predicate

Computation of oriented bounding box for cloud of points.

yade._packObb.**cloudBestFitOBB**(*(tuple)arg1*) → tuple
Return (Vector3 center, Vector3 halfSize, Quaternion orientation) of best-fit oriented bounding-box for given tuple of points (uses brute-force velome minimization, do not use for very large clouds).

2.4.13 yade.plot module

Module containing utility functions for plotting inside yade. See examples/simple-scene/simple-scene-plot.py or examples/concrete/uniax.py for example of usage.

yade.plot.**addAutoData**()
Add data by evaluating contents of *plot.plots*. Expressions rasing exceptions will be handled gracefully, but warning is printed for each.

```
>>> from yade import plot
>>> from pprint import pprint
>>> O.reset()
>>> plot.resetData()
>>> plot.plots={'O.iter':('O.time',None,'numParticles=len(O.bodies)')}
>>> plot.addAutoData()
>>> pprint(plot.data)
{'O.iter': [0], 'O.time': [0.0], 'numParticles': [0]}
```

Note that each item in *plot.plots* can be

- an expression to be evaluated (using the `eval` builtin);
- `name=expression` string, where `name` will appear as label in plots, and expression will be evaluated each time;
- a dictionary-like object – current keys are labels of plots and current values are added to *plot.data*. The contents of the dictionary can change over time, in which case new lines will be created as necessary.

A simple simulation with plot can be written in the following way; note how the energy plot is specified.

```
>>> from yade import plot, utils
>>> from yade.minieigenHP import *  # not needed in actual yade session, but␣
↪needed for doctests
>>> from yade.wrapper import * # not needed in actual yade session, but needed␣
↪for doctests
>>> plot.plots={'i=O.iter':(O.energy,None,'total energy=O.energy.total()')}
>>> # we create a simple simulation with one ball falling down
>>> plot.resetData()
>>> O.bodies.append(utils.sphere((0,0,0),1))
0
>>> O.dt=utils.PWaveTimeStep()
>>> O.engines=[
...    ForceResetter(),
...    GravityEngine(gravity=(0,0,-10),warnOnce=False),
...    NewtonIntegrator(damping=.4,kinSplit=True),
...    # get data required by plots at every step
...    PyRunner(command='yade.plot.addAutoData()',iterPeriod=1,initRun=True)
... ]
>>> O.trackEnergy=True
>>> O.run(2,True)
>>> pprint(plot.data)
{'gravWork': [0.0, -25.13274...],
 'i': [0, 1],
 'kinRot': [0.0, 0.0],
 'kinTrans': [0.0, 7.5398...],
 'nonviscDamp': [0.0, 10.0530...],
 'total energy': [0.0, -7.5398...]}
```

`yade.plot.addData`(**d_in*, ***kw*)

Add data from arguments name1=value1,name2=value2 to yade.plot.data. (the old {'name1':value1,'name2':value2} is deprecated, but still supported)

New data will be padded with nan's, unspecified data will be nan (nan's don't appear in graphs). This way, equal length of all data is assured so that they can be plotted one against any other.

```
>>> from yade import plot
>>> from yade.minieigenHP import *  # not needed in actual yade session, but␣
↪needed for doctests
>>> from pprint import pprint
>>> plot.resetData()
>>> plot.addData(a=1)
>>> plot.addData(b=2)
>>> plot.addData(a=3,b=4)
>>> pprint(plot.data)
{'a': [1, nan, 3], 'b': [nan, 2, 4]}
```

Some sequence types can be given to addData; they will be saved in synthesized columns for individual components.

```
>>> plot.resetData()
>>> plot.addData(c=Vector3(5,6,7),d=Matrix3(8,9,10, 11,12,13, 14,15,16))
>>> pprint(plot.data)
{'c_x': [5.0],
 'c_y': [6.0],
 'c_z': [7.0],
 'd_xx': [8.0],
 'd_xy': [9.0],
 'd_xz': [10.0],
 'd_yx': [11.0],
 'd_yy': [12.0],
 'd_yz': [13.0],
 'd_zx': [14.0],
 'd_zy': [15.0],
 'd_zz': [16.0]}
```

yade.plot.autozoom = True

Enable/disable automatic plot rezooming after data update. Sometimes rezooming must be skipped unless a call to *plot.setLiveForceAlwaysUpdate* forces it to work.

yade.plot.data = {'eps': [0.0001, 0.001, nan], 'force': [nan, nan, 1000.0], 'sigma': [12, nan, nan]}

Global dictionary containing all data values, common for all plots, in the form {'name':[value,...],...}. Data should be added using plot.addData function. All [value,...] columns have the same length, they are padded with NaN if unspecified.

yade.plot.labels = {}

Dictionary converting names in data to human-readable names (TeX names, for instance); if a variable is not specified, it is left untranslated.

yade.plot.live = False

Enable/disable live plot updating.

yade.plot.liveInterval = 1

Interval for the live plot updating, in seconds.

yade.plot.plot(*noShow=False*, *subPlots=True*)

Do the actual plot, which is either shown on screen (and nothing is returned: if *noShow* is `False` - note that your yade compilation should present qt4 feature so that figures can be displayed) or, if *noShow* is `True`, returned as matplotlib's Figure object or list of them.

You can use

```
>>> from yade import plot
>>> plot.resetData()
>>> plot.plots={'foo':('bar',)}
>>> plot.plot(noShow=True).savefig('someFile.pdf')
>>> import os
>>> os.path.exists('someFile.pdf')
True
>>> os.remove('someFile.pdf')
```

to save the figure to file automatically.

Note

For backwards compatibility reasons, *noShow* option will return list of figures for multiple figures but a single figure (rather than list with 1 element) if there is only 1 figure.

`yade.plot.plots = {'i': ('t',), 'i ': ('z1', 'v1')}`

dictionary x-name -> (yspec,...), where yspec is either y-name or (y-name,'line-specification'). If `(yspec,...)` is `None`, then the plot has meaning of image, which will be taken from respective field of *plot.imgData*.

`yade.plot.reset()`

Reset all plot-related variables (data, plots, labels)

`yade.plot.resetData()`

Reset all plot data; keep plots and labels intact.

`yade.plot.reverseData()`

Reverse yade.plot.data order.

Useful for tension-compression test, where the initial (zero) state is loaded and, to make data continuous, last part must *end* in the zero state.

`yade.plot.saveDataTxt(`*fileName*, *vars=None*, *headers=None*`)`

Save plot data into a (optionally compressed) text file. The first line contains a comment (starting with `#`) giving variable name for each of the columns. This format is suitable for being loaded for further processing (outside yade) with `numpy.genfromtxt` function, which recognizes those variable names (creating numpy array with named entries) and handles decompression transparently.

```
>>> from yade import plot
>>> from pprint import pprint
>>> plot.reset()
>>> plot.addData(a=1,b=11,c=21,d=31)  # add some data here
>>> plot.addData(a=2,b=12,c=22,d=32)
>>> pprint(plot.data)
{'a': [1, 2], 'b': [11, 12], 'c': [21, 22], 'd': [31, 32]}
>>> plot.saveDataTxt('/tmp/dataFile.txt.tar.gz',vars=('a','b','c'))
>>> import numpy
>>> d=numpy.genfromtxt('/tmp/dataFile.txt.tar.gz',dtype=None,names=True)
>>> d['a']
array([1, 2])
>>> d['b']
array([11, 12])
>>> import os # cleanup
>>> os.remove('/tmp/dataFile.txt.tar.gz')
```

Parameters

- **`fileName`** – file to save data to; if it ends with `.bz2` / `.gz`, the file will be compressed using bzip2 / gzip.
- **`vars`** – Sequence (tuple/list/set) of variable names to be saved. If `None` (default), all variables in *plot.plot* are saved.
- **`headers`** – Set of parameters to write on header

`yade.plot.saveGnuplot(`*baseName*, *term='wxt'*, *extension=None*, *timestamp=False*, *comment=None*, *title=None*, *varData=False*`)`

Save data added with *plot.addData* into (compressed) file and create .gnuplot file that attempts to mimick plots specified with *plot.plots*.

Parameters

- **baseName** – used for creating baseName.gnuplot (command file for gnuplot), associated `baseName.data.bz2` (data) and output files (if applicable) in the form `baseName.[plot number].extension`
- **term** – specify the gnuplot terminal; defaults to `x11`, in which case gnuplot will draw persistent windows to screen and terminate; other useful terminals are `png`, `cairopdf` and so on
- **extension** – extension for `baseName` defaults to terminal name; fine for png for example; if you use `cairopdf`, you should also say `extension='pdf'` however
- **timestamp** (*bool*) – append numeric time to the basename
- **varData** (*bool*) – whether file to plot will be declared as variable or be in-place in the plot expression
- **comment** – a user comment (may be multiline) that will be embedded in the control file

Returns

name of the gnuplot file created.

yade.plot.**savePlotSequence**(*fileBase*, *stride=1*, *imgRatio=(5, 7)*, *title=None*, *titleFrames=20*, *lastFrames=30*)

Save sequence of plots, each plot corresponding to one line in history. It is especially meant to be used for *utils.makeVideo*.

Parameters

- **stride** – only consider every stride-th line of history (default creates one frame per each line)
- **title** – Create title frame, where lines of title are separated with newlines (`\n`) and optional subtitle is separated from title by double newline.
- **titleFrames** (*int*) – Create this number of frames with title (by repeating its filename), determines how long the title will stand in the movie.
- **lastFrames** (*int*) – Repeat the last frame this number of times, so that the movie does not end abruptly.

Returns

List of filenames with consecutive frames.

yade.plot.**setLiveForceAlwaysUpdate**(*forceLiveUpdate*)

The *plot.liveInterval* and *plot.live* control live refreshing of the plot during calculations. The refreshing is done in a separate thread, so that it does not interfere with calculations. Drawing the data will not work when at exactly the same time it is being updated in other thread. Use `yade.plot.setLiveForceAlwaysUpdate(True)` if you want calculations to **PAUSE** during the plot updates. This function returns current `bool` value of forced updates if the call was a success, otherwise it returns a `str` with explanation why it failed. It is guaranteed to work if simulation was paused with *O.pause()* call.

yade.plot.**splitData**()

Make all plots discontinuous at this point (adds nan's to all data fields)

2.4.14 yade.polyhedra_utils module

Auxiliary functions for polyhedra

yade.polyhedra_utils.**fillBox**(*mincoord*, *maxcoord*, *material*, *sizemin=[1, 1, 1]*, *sizemax=[1, 1, 1]*, *ratio=[0, 0, 0]*, *seed=None*, *mask=1*)

fill box [mincoord, maxcoord] by non-overlaping polyhedrons with random geometry and sizes within the range (uniformly distributed) :param Vector3 mincoord: first corner :param Vector3

maxcoord: second corner :param Vector3 sizemin: minimal size of bodies :param Vector3 sizemax: maximal size of bodies :param Vector3 ratio: scaling ratio :param float seed: random seed

yade.polyhedra_utils.**fillBoxByBalls**(*mincoord*, *maxcoord*, *material*, *sizemin=[1, 1, 1]*, *sizemax=[1, 1, 1]*, *ratio=[0, 0, 0]*, *seed=None*, *mask=1*, *numpoints=60*)

yade.polyhedra_utils.**polyhedra**(*material*, *size=Vector3(1, 1, 1)*, *seed=None*, *v=[]*, *mask=1*, *fixed=False*, *color=[-1, -1, -1]*)

create polyhedra, one can specify vertices directly, or leave it empty for random shape.

Parameters

- **material** (*Material*) – material of new body
- **size** (Vector3) – size of new body (see Polyhedra docs)
- **seed** (*float*) – seed for random operations
- **v** (*[*Vector3*]*) – list of body vertices (see Polyhedra docs)

yade.polyhedra_utils.**polyhedraSnubCube**(*radius*, *material*, *centre*, *mask=1*)

yade.polyhedra_utils.**polyhedraTruncIcosaHed**(*radius*, *material*, *centre*, *mask=1*)

yade.polyhedra_utils.**polyhedralBall**(*radius*, *N*, *material*, *center*, *mask=1*)

creates polyhedra having N vertices and resembling sphere

Parameters

- **radius** (*float*) – ball radius
- **N** (*int*) – number of vertices
- **material** (*Material*) – material of new body
- **center** (Vector3) – center of the new body

yade._polyhedra_utils.**MaxCoord**(*(yade.wrapper.Shape)arg1*, *(yade.wrapper.State)arg2*) → *yade._minieigenHP.Vector3*

returns max coordinates

yade._polyhedra_utils.**MinCoord**(*(yade.wrapper.Shape)arg1*, *(yade.wrapper.State)arg2*) → *yade._minieigenHP.Vector3*

returns min coordinates

yade._polyhedra_utils.**PrintPolyhedra**(*(yade.wrapper.Shape)arg1*) → None

Print list of vertices sorted according to polyhedrons facets.

yade._polyhedra_utils.**PrintPolyhedraActualPos**(*(yade.wrapper.Shape)arg1*, *(yade.wrapper.State)arg2*) → None

Print list of vertices sorted according to polyhedrons facets.

yade._polyhedra_utils.**SieveCurve**() → None

save sieve curve coordinates into file

yade._polyhedra_utils.**SieveSize**(*(yade.wrapper.Shape)arg1*) → float

returns approximate sieve size of polyhedron

yade._polyhedra_utils.**SizeOfPolyhedra**(*(yade.wrapper.Shape)arg1*) → *yade._minieigenHP.Vector3*

returns max, middle an min size in perpendicular directions

yade._polyhedra_utils.**SizeRatio**() → None

save sizes of polyhedra into file

yade._polyhedra_utils.**Split**(*(yade.wrapper.Body)arg1*, *(yade._minieigenHP.Vector3)arg2*, *(yade._minieigenHP.Vector3)arg3*) → None

split polyhedron perpendicularly to given direction through given point

yade._polyhedra_utils.**convexHull**(*(object)arg1*) → bool

TODO

yade._polyhedra_utils.**do_Polyhedras_Intersect**(*(yade.wrapper.Shape)arg1*, *(yade.wrapper.Shape)arg2*, *(yade.wrapper.State)arg3*, *(yade.wrapper.State)arg4*) → bool

check polyhedras intersection

yade._polyhedra_utils.**fillBoxByBalls_cpp**(*(yade._minieigenHP.Vector3)arg1*, *(yade._minieigenHP.Vector3)arg2*, *(yade._minieigenHP.Vector3)arg3*, *(yade._minieigenHP.Vector3)arg4*, *(yade._minieigenHP.Vector3)arg5*, *(int)arg6*, *(yade.wrapper.Material)arg7*, *(int)arg8*) → object

Generate non-overlaping 'spherical' polyhedrons in box

yade._polyhedra_utils.**fillBox_cpp**(*(yade._minieigenHP.Vector3)arg1*, *(yade._minieigenHP.Vector3)arg2*, *(yade._minieigenHP.Vector3)arg3*, *(yade._minieigenHP.Vector3)arg4*, *(yade._minieigenHP.Vector3)arg5*, *(int)arg6*, *(yade.wrapper.Material)arg7*) → object

Generate non-overlaping polyhedrons in box

2.4.15 yade.post2d module

Module for 2d postprocessing, containing classes to project points from 3d to 2d in various ways, providing basic but flexible framework for extracting arbitrary scalar values from bodies/interactions and plotting the results. There are 2 basic components: flatteners and extractors.

The algorithms operate on bodies (default) or interactions, depending on the `intr` parameter of post2d.data.

Flatteners

Instance of classes that convert 3d (model) coordinates to 2d (plot) coordinates. Their interface is defined by the *post2d.Flatten* class (`__call__`, `planar`, `normal`).

Extractors

Callable objects returning scalar or vector value, given a body/interaction object. If a 3d vector is returned, Flattener.planar is called, which should return only in-plane components of the vector.

Example

This example can be found in examples/concrete/uniax-post.py

```
from yade import post2d
import pylab # the matlab-like interface of matplotlib

O.load('/tmp/uniax-tension.xml.bz2')

# flattener that project to the xz plane
flattener=post2d.AxisFlatten(useRef=False,axis=1)
# return scalar given a Body instance
```

(continues on next page)

(continued from previous page)

```
extractDmg=lambda b: b.state.normDmg
# will call flattener.planar implicitly
# the same as: extractVelocity=lambda b: flattener.planar(b,b.state.vel)
extractVelocity=lambda b: b.state.vel

# create new figure
pylab.figure()
# plot raw damage
post2d.plot(post2d.data(extractDmg,flattener))

# plot smooth damage into new figure
pylab.figure(); ax,map=post2d.plot(post2d.data(extractDmg,flattener,stDev=2e-3))
# show color scale
pylab.colorbar(map,orientation='horizontal')

# raw velocity (vector field) plot
pylab.figure(); post2d.plot(post2d.data(extractVelocity,flattener))

# smooth velocity plot; data are sampled at regular grid
pylab.figure(); ax,map=post2d.plot(post2d.data(extractVelocity,flattener,stDev=1e-3))
# save last (current) figure to file
pylab.gcf().savefig('/tmp/foo.png')

# show the figures
pylab.show()
```

class yade.post2d.**AxisFlatten**(*inherits Flatten → object*)

__init__(*useRef=False*, *axis=2*)

Parameters

- **useRef** (*bool*) – use reference positions rather than actual positions (only meaningful when operating on Bodies)
- **axis** (*{0,1,2}*) – axis normal to the plane; the return value will be simply position with this component dropped.

normal(*pos*, *vec*)

Given position and vector value, return lenght of the vector normal to the flat plane.

planar(*pos*, *vec*)

Given position and vector value, project the vector value to the flat plane and return its 2 in-plane components.

class yade.post2d.**CylinderFlatten**(*inherits Flatten → object*)

Class for converting 3d point to 2d based on projection onto plane from circle. The y-axis in the projection corresponds to the rotation axis; the x-axis is distance form the axis.

__init__(*useRef*, *axis=2*)

Parameters

- **useRef** – (bool) use reference positions rather than actual positions
- **axis** – axis of the cylinder, {0,1,2}

normal(*b*, *vec*)

Given position and vector value, return lenght of the vector normal to the flat plane.

`planar`(*b*, *vec*)

Given position and vector value, project the vector value to the flat plane and return its 2 in-plane components.

`class yade.post2d.Flatten`(*inherits object*)

Abstract class for converting 3d point into 2d. Used by post2d.data2d.

`normal`(*pos*, *vec*)

Given position and vector value, return lenght of the vector normal to the flat plane.

`planar`(*pos*, *vec*)

Given position and vector value, project the vector value to the flat plane and return its 2 in-plane components.

`class yade.post2d.HelixFlatten`(*inherits Flatten → object*)

Class converting 3d point to 2d based on projection from helix. The y-axis in the projection corresponds to the rotation axis

`__init__`(*useRef*, *thetaRange*, *dH_dTheta*, *axis=2*, *periodStart=0*)

Parameters

- **useRef** (*bool*) – use reference positions rather than actual positions
- **thetaRange** (*(min, max)*) – bodies outside this range will be discarded
- **dH_dTheta** (*float*) – inclination of the spiral (per radian)
- **axis** (*{0,1,2}*) – axis of rotation of the spiral
- **periodStart** (*float*) – height of the spiral for zero angle

`normal`(*pos*, *vec*)

Given position and vector value, return lenght of the vector normal to the flat plane.

`planar`(*b*, *vec*)

Given position and vector value, project the vector value to the flat plane and return its 2 in-plane components.

`yade.post2d.data`(*extractor*, *flattener*, *intr=False*, *onlyDynamic=True*, *stDev=None*, *relThreshold=3.0*, *perArea=0*, *div=(50, 50)*, *margin=(0, 0)*, *radius=1*)

Filter all bodies/interactions, project them to 2d and extract required scalar value; return either discrete array of positions and values, or smoothed data, depending on whether the stDev value is specified.

The `intr` parameter determines whether we operate on bodies or interactions; the extractor provided should expect to receive body/interaction.

Parameters

- **extractor** (*callable*) – receives *Body* (or *Interaction*, if `intr` is `True`) instance, should return scalar, a 2-tuple (vector fields) or None (to skip that body/interaction)
- **flattener** (*callable*) – *post2d.Flatten* instance, receiving body/interaction, returns its 2d coordinates or `None` (to skip that body/interaction)
- **intr** (*bool*) – operate on interactions rather than bodies
- **onlyDynamic** (*bool*) – skip all non-dynamic bodies
- **stDev** (*float/None*) – standard deviation for averaging, enables smoothing; `None` (default) means raw mode, where discrete points are returned
- **relThreshold** (*float*) – threshold for the gaussian weight function relative to stDev (smooth mode only)

- **perArea** (*int*) – if 1, compute weightedSum/weightedArea rather than weighted average (weightedSum/sumWeights); the first is useful to compute average stress; if 2, compute averages on subdivision elements, not using weight function
- **div** (*(int,int)*) – number of cells for the gaussian grid (smooth mode only)
- **margin** (*(float,float)*) – x,y margins around bounding box for data (smooth mode only)
- **radius** (*float/callable*) – Fallback value for radius (for raw plotting) for non-spherical bodies or interactions; if a callable, receives body/interaction and returns radius

Returns
dictionary

Returned dictionary always containing keys 'type' (one of 'rawScalar','rawVector','smoothScalar','smoothVector', depending on value of smooth and on return value from extractor), 'x', 'y', 'bbox'.

Raw data further contains 'radii'.

Scalar fields contain 'val' (value from *extractor*), vector fields have 'valX' and 'valY' (2 components returned by the *extractor*).

yade.post2d.plot(*data*, *axes=None*, *alpha=0.5*, *clabel=True*, *cbar=False*, *aspect='equal'*, ***kw*)

Given output from post2d.data, plot the scalar as discrete or smooth plot.

For raw discrete data, plot filled circles with radii of particles, colored by the scalar value.

For smooth discrete data, plot image with optional contours and contour labels.

For vector data (raw or smooth), plot quiver (vector field), with arrows colored by the magnitude.

Parameters

- **axes** – matplotlib.axesinstance where the figure will be plotted; if None, will be created from scratch.
- **data** – value returned by *post2d.data*
- **clabel** (*bool*) – show contour labels (smooth mode only), or annotate cells with numbers inside (with perArea==2)
- **cbar** (*bool*) – show colorbar (equivalent to calling pylab.colorbar(mappable) on the returned mappable)

Returns
tuple of `(axes,mappable)`; mappable can be used in further calls to pylab.colorbar.

2.4.16 yade.potential_utils module

Auxiliary functions for the Potential Blocks

yade.potential_utils.aabbPlates(*material*, *extrema=None*, *thickness=0.0*, *r=0.0*, *R=0.0*, *mask=1*, *isBoundary=False*, *fixed=True*, *color=None*)

Return 6 cuboids that will wrap existing packing as walls from all sides. #FIXME: Correct this comment

Parameters

- **material** (*Material*) – material of new bodies (FrictMat)
- **extrema** (*[Vector3, Vector3]*) – extremal points of the Aabb of the packing, as a list of two Vector3, or any equivalent type (will not be calculated if not specified)
- **thickness** (*float*) – wall thickness (equal to 1/10 of the smallest dimension if not specified)

- **r** (*float*) – radius of inner Potential Particle (see PotentialBlock docs)
- **R** (*float*) – distance R of the Potential Blocks (see PotentialBlock docs)
- **mask** (*int*) – groupMask for the new bodies

Returns
a list of 6 PotentialBlock Bodies enclosing the packing, in the order minX,maxX,minY,maxY,minZ,maxZ.

yade.potential_utils.**cuboid**(*material*, *edges=Vector3(1, 1, 1)*, *r=0.0*, *R=0.0*, *center=[0, 0, 0]*, *mask=1*, *isBoundary=False*, *fixed=False*, *color=[-1, -1, -1]*)

creates cuboid using the Potential Blocks

Parameters

- **edges** (Vector3) – edges of the cuboid
- **material** (*Material*) – material of new body (FrictMat)
- **center** (Vector3) – center of the new body

yade.potential_utils.**cylindricalPlates**(*material*, *radius=0.0*, *height=0.0*, *thickness=0.0*, *numFaces=3*, *r=0.0*, *R=0.0*, *mask=1*, *isBoundary=False*, *fixed=True*, *lid=[True, True]*, *color=None*)

Return numFaces cuboids that will wrap existing packing as walls from all sides. #FIXME: Correct this comment

Parameters

- **material** (*Material*) – material of new bodies (FrictMat)
- **radius** (*float*) – radius of the cylinder
- **height** (*float*) – height of cylinder
- **thickness** (*float*) – thickness of cylinder faces (equal to 1/10 of the cylinder inradius if not specified)
- **numFaces** (*int*) – number of cylinder faces (>3)
- **r** (*float*) – radius of inner Potential Particle (see PotentialBlock docs)
- **R** (*float*) – distance R of the Potential Blocks (see PotentialBlock docs)
- **mask** (*int*) – groupMask for the new bodies
- **[bool]** (*lid*) – list of booleans, whether to create the bottom and top lids of the cylindrical plates

Returns
a list of cuboidal Potential Blocks forming a hollow cylinder

yade.potential_utils.**platonic_solid**(*material*, *numFaces*, *edge=0.0*, *ri=0.0*, *rm=0.0*, *rc=0.0*, *volume=0.0*, *r=0.0*, *R=None*, *center=[0, 0, 0]*, *mask=1*, *isBoundary=False*, *fixed=False*, *color=[-1, -1, -1]*)

yade.potential_utils.**potentialblock**(*material*, *a=[]*, *b=[]*, *c=[]*, *d=[]*, *r=0.0*, *R=0.0*, *mask=1*, *isBoundary=False*, *fixed=False*, *color=[-1, -1, -1]*)

creates potential block.

Parameters

- **material** (*Material*) – material of new body
- **a,b,c,d** (*[float]*) – lists of plane coefficients of the particle faces (see PotentialBlock docs)
- **r** (*float*) – radius of inner Potential Particle (see PotentialBlock docs)

- **R** (*float*) – distance R of the Potential Blocks (see PotentialBlock docs)
- **isBoundary** (*bool*) – whether this is a boundary body (see PotentialBlock docs)

yade.potential_utils.**prism**(*material*, *radius1=0.0*, *radius2=-1*, *thickness=0.0*, *numFaces=3*, *r=0.0*, *R=0.0*, *center=None*, *color=[1, 0, 0]*, *mask=1*, *isBoundary=False*, *fixed=False*)

Return regular prism with numFaces

Parameters

- **material** (*Material*) – material of new bodies (FrictMat)
- **radius1** (*float*) – inradius of the start cross-section of the prism
- **radius2** (*float*) – inradius of the end cross-section of the prism (equal to radius1 if not specified)
- **thickness** (*float*) – thickness of the prism (equal to radius1 if not specified)
- **numFaces** (*int*) – number of prisms' faces (>3)
- **r** (*float*) – radius of inner Potential Particle (see PotentialBlock docs)
- **R** (*float*) – distance R of the Potential Blocks (see PotentialBlock docs)
- **center** (Vector3) – center of the Potential Blocks (not currently used)
- **mask** (*int*) – groupMask for the new bodies

Returns

an axial-symmetric Potential Block with variable cross-section, which can become either a regular prism (radius1=radius2), a pyramid (radius2=0) or a cylinder or cone respectively, for a large enough numFaces value.

2.4.17 yade.qt module

2.4.18 yade.timing module

Functions for accessing timing information stored in engines and functors.

See *Timing* section of the programmer's manual for some examples (https://yade-dem.org/doc/prog.html#timing).

yade.timing.**reset**()

Zero all timing data.

yade.timing.**runtime**()

Return total running time (same as last line in the output of stats()) in nanoseconds

yade.timing.**stats**()

Print summary table of timing information from engines and functors. Absolute times as well as percentages are given. Sample output:

```
Name                                                    Count                ␣
↪Time            Rel. time
-------------------------------------------------------------------------------
↪----------------------
ForceResetter                                             102                2150us     ␣
↪           0.02%
"collider"                                                  5               64200us     ␣
↪           0.60%
InteractionLoop                                           102            10571887us     ␣
↪          98.49%
"combEngine"                                              102                8362us     ␣
```

(continues on next page)

(continued from previous page)

```
↪           0.08%
"newton"                                          102              73166us    ␣
↪           0.68%
"cpmStateUpdater"                                   1               9605us    ␣
↪           0.09%
PyRunner                                            1                136us    ␣
↪           0.00%
"plotDataCollector"                                 1                291us    ␣
↪           0.00%
TOTAL                                                          10733564us    ␣
↪         100.00%
```

sample output (compiled with -DENABLE_PROFILING=1 option):

```
Name                                                    Count                ␣
↪Time            Rel. time
--------------------------------------------------------------------------------
↪----------------------
ForceResetter                                     102               2150us    ␣
↪           0.02%
"collider"                                          5              64200us    ␣
↪           0.60%
InteractionLoop                                   102           10571887us    ␣
↪          98.49%
  Ig2_Sphere_Sphere_ScGeom                      1222186            1723168us  ␣
↪            16.30%
    Ig2_Sphere_Sphere_ScGeom                      1222186            1723168us␣
↪             100.00%
  Ig2_Facet_Sphere_ScGeom                           753               1157us  ␣
↪             0.01%
    Ig2_Facet_Sphere_ScGeom                           753               1157us␣
↪             100.00%
  Ip2_CpmMat_CpmMat_CpmPhys                       11712              26015us  ␣
↪             0.25%
    end of Ip2_CpmPhys                              11712              26015us␣
↪             100.00%
  Ip2_FrictMat_CpmMat_FrictPhys                       0                  0us  ␣
↪             0.00%
  Law2_ScGeom_CpmPhys_Cpm                       3583872            4819289us  ␣
↪            45.59%
    GO A                                          1194624            1423738us␣
↪              29.54%
    GO B                                          1194624            1801250us␣
↪              37.38%
    rest                                          1194624            1594300us␣
↪              33.08%
    TOTAL                                         3583872            4819289us␣
↪             100.00%
  Law2_ScGeom_FrictPhys_CundallStrack                 0                  0us  ␣
↪             0.00%
"combEngine"                                      102               8362us    ␣
↪           0.08%
"newton"                                          102              73166us    ␣
↪           0.68%
"cpmStateUpdater"                                   1               9605us    ␣
↪           0.09%
```

(continues on next page)

(continued from previous page)

```
PyRunner                                                  1                 136us
↪            0.00%
"plotDataCollector"                                       1                 291us
↪            0.00%
TOTAL                                                                   10733564us
↪          100.00%
```

2.4.19 yade.utils module

Heap of functions that don't (yet) fit anywhere else.

Devs: please DO NOT ADD more functions here, it is getting too crowded!

`yade.utils.NormalRestitution2DampingRate(en)`

Compute the normal damping rate as a function of the normal coefficient of restitution e_n. For $e_n \in \langle 0, 1 \rangle$ damping rate equals

$$-\frac{\log e_n}{\sqrt{e_n^2 + \pi^2}}$$

`yade.utils.SpherePWaveTimeStep(radius, density, young)`

Compute P-wave critical timestep for a single (presumably representative) sphere, using formula for P-Wave propagation speed $\Delta t_c = \frac{r}{\sqrt{E/\rho}}$. If you want to compute minimum critical timestep for all spheres in the simulation, use *utils.PWaveTimeStep* instead.

```
>>> SpherePWaveTimeStep(1e-3,2400,30e9)
2.8284271247461903e-07
```

`class yade.utils.TableParamReader(inherits object)`

Class for reading simulation parameters from text file.

Each parameter is represented by one column, each parameter set by one line. Colums are separated by blanks (no quoting).

First non-empty line contains column titles (without quotes). You may use special column named 'description' to describe this parameter set; if such colum is absent, description will be built by concatenating column names and corresponding values (`param1=34,param2=12.22,param4=foo`)

- from columns ending in `!` (the `!` is not included in the column name)
- from all columns, if no columns end in `!`.

Empty lines within the file are ignored (although counted); `#` starts comment till the end of line. Number of blank-separated columns must be the same for all non-empty lines.

A special value `=` can be used instead of parameter value; value from the previous non-empty line will be used instead (works recursively).

This class is used by *utils.readParamsFromTable*.

`__init__(file)`

Setup the reader class, read data into memory.

`paramDict()`

Return dictionary containing data from file given to constructor. Keys are line numbers (which might be non-contiguous and refer to real line numbers that one can see in text editors), values are dictionaries mapping parameter names to their values given in the file. The special value '=' has already been interpreted, `!` (bangs) (if any) were already removed from column titles, `description` column has already been added (if absent).

class yade.utils.YadeColorStyle(*inherits object*)

Parameters for default colors and 3D view parameters. Switch between styles with *colorStyle.setStyle("styleName")*. See also the *rendering section* of user manual.

__init__(*bgColor=(0.2, 0.2, 0.2)*, *rgbMin=Vector3(0.4050000000000000266, 0.3599999999999999867, 0.1350000000000000089)*, *rgbRange=Vector3(0.2399999999999999911, 0.2399999999999999911, 0.3599999999999999867)*, *uniScale=False*, *wallColor=Vector3(0.8000000000000000444, 0.8000000000000000444, 0.5999999999999999778)*, *stripes=True*, *quality=1*)

Generic data class for defining color styles. rgbRange is the length of interval [min,max] for each (rgb) color. If uniScale=True the random color is rgbMin+random*rgbRange, else each component is generated randomly.

applyAll(*ids=None*)

applyBodyStyle(*ids=None*)

randomColor()

Generate a random RGB color between rgbMin and rgbMax.

setBackgroundColor()

yade.utils.aabbDim(*cutoff=0.0*, *centers=False*)

Return dimensions of the axis-aligned bounding box, optionally with relative part *cutoff* cut away.

yade.utils.aabbExtrema2d(*pts*)

Return 2d bounding box for a sequence of 2-tuples.

yade.utils.aabbWalls(*extrema=None*, *thickness=0*, *oversizeFactor=1.5*, ***kw*)

Return 6 boxes that will wrap existing packing as walls from all sides.

Parameters

- **extrema** – extremal points of the Aabb of the packing, as a list of two Vector3, or any equivalent type (will be calculated if not specified)
- **thickness** (*float*) – is wall thickness (will be 1/10 of the X-dimension if not specified)
- **oversizeFactor** (*float*) – factor to enlarge walls in their plane.

Returns

a list of 6 wall Bodies enclosing the packing, in the order minX,maxX,minY,maxY,minZ,maxZ.

yade.utils.avgNumInteractions(*cutoff=0.0*, *skipFree=False*, *considerClumps=False*)

Return average numer of interactions per particle, also known as *coordination number* Z. This number is defined as

$$Z = 2C/N$$

where C is number of contacts and N is number of particles. When clumps are present, number of particles is the sum of standalone spheres plus the sum of clumps. Clumps are considered in the calculation if cutoff != 0 or skipFree = True. If cutoff=0 (default) and skipFree=False (default) one needs to set considerClumps=True to consider clumps in the calculation.

With *skipFree*, particles not contributing to stable state of the packing are skipped, following equation (8) given in [Thornton2000]:

$$Z_m = \frac{2C - N_1}{N - N_0 - N_1}$$

Parameters

- **cutoff** – cut some relative part of the sample's bounding box away.
- **skipFree** – see above.
- **considerClumps** – also consider clumps if cutoff=0 and skipFree=False; for further explanation see above.

yade.utils.box(*center*, *extents*, *orientation=Quaternion((1, 0, 0), 0)*, *dynamic=None*, *fixed=False*, *wire=False*, *color=Vector3(0.80000000000000444, 0.80000000000000444, 0.59999999999999778)*, *highlight=False*, *material=-1*, *mask=1*)

Create box (cuboid) with given parameters.

Parameters

- **extents** (Vector3) – half-sizes along x,y,z axes. Use can be made of *orientation* parameter in case those box-related axes do not conform the simulation axes
- **orientation** (Quaternion) – assigned to the *body's orientation*, which corresponds to rotating the *extents* axes

See *utils.sphere*'s documentation for meaning of other parameters.

class yade.utils.clumpTemplate(*inherits object*)

Create a clump template by a list of relative radii and a list of relative positions. Both lists must have the same length.

Parameters

- **relRadii** (*[float,float,...]*) – list of relative radii (minimum length = 2)
- **relPositions** (*[Vector3,Vector3,...]*) – list of relative positions (minimum length = 2)

yade.utils.defaultMaterial()

Return default material, when creating bodies with *utils.sphere* and friends, material is unspecified and there is no shared material defined yet. By default, this function returns

```
FrictMat(density=1e3,young=1e7,poisson=.3,frictionAngle=.5,label='defaultMat')
```

yade.utils.facet(*vertices*, *dynamic=None*, *fixed=True*, *wire=True*, *color=Vector3(0.80000000000000444, 0.80000000000000444, 0.59999999999999778)*, *highlight=False*, *noBound=False*, *material=-1*, *mask=1*)

Create a *Facet*-shaped body with given parameters. Body center is chosen as the center of the inscribed circle of the *vertices* triangle

Parameters

- **vertices** (*[Vector3,Vector3,Vector3]*) – coordinates of vertices in the global coordinate system.
- **wire** (*bool*) – if True, facets are shown as skeleton; otherwise facets are filled
- **noBound** (*bool*) – set *Body.bounded*
- **color** (*Vector3-or-None*) – color of the facet; random color will be assigned if None.

See *utils.sphere*'s documentation for meaning of other parameters.

yade.utils.fractionalBox(*fraction=1.0*, *minMax=None*)

Return (min,max) that is the original minMax box (or aabb of the whole simulation if not specified) linearly scaled around its center to the fraction factor

yade.utils.**levelSetBody**(*shape=''*, *center=Vector3(0, 0, 0)*, *radius=0*, *extents=Vector3(0, 0, 0)*, *epsilons=Vector2(0, 0)*, *clump=None*, *spacing=0.1*, *grid=None*, *distField=[]*, *smearCoeff=1.5*, *nSurfNodes=102*, *surfNodes=[]*, *nodesPath=2*, *nodesTol=50*, *n_neighborsNodes=-1*, *orientation=Quaternion((1, 0, 0), 0)*, *hasAABE=False*, *axesAABE=Vector3(0, 0, 0)*, *dynamic=True*, *material=-1*, *starLike=False*, *color=Vector3(1, 1, 1)*)

Creates a *LevelSet* shaped body through various workflows: one can choose to define the discrete distance field from pre-defined shapes (through *shape* and related arguments), or to mimick a *Clump* instance (*clump* argument, for comparison purposes), or directly assign the discrete distance field on some grid (*distField* and *grid* arguments). Surface nodes can also be either ray traced (see *nSurfNodes*, *nodesPath* and *nodesTol*) or directly assigned (see *surfNodes*)

Parameters

- **shape** (*string*) – use this argument to enjoy predefined shapes among 'sphere', 'box' (for a rectangular parallelepiped), 'disk' (for a 2D analysis in (x,y) plane), or 'superellipsoid'; in conjunction with *extents* or *radius* attributes. Superellipsoid surfaces are defined in local axes (inertial frame) by the following equation: $f(x,y,z) = (|x/r_x|^{2/\varepsilon_e} + |y/r_y|^{2/\varepsilon_e})^{\varepsilon_e/\varepsilon_n} + |z/r_z|^{2/\varepsilon_n} = 1$ and their distance field is obtained thanks to a *Fast Marching Method*.
- **center** (Vector3) – (initial) position of that body
- **clump** (*Clump*) – pass here a multi-sphere (other cases of Clump not supported) instance to mimick, if desired
- **radius** (Real) – imposed radius in case *shape* = 'sphere' or 'disk'
- **extents** (Vector3) – half extents along the local axes in case *shape* = 'box' or 'superellipsoid' (r_x, r_y, r_z for the latter)
- **epsilons** (Vector2) – in case *shape* = 'superellipsoid', the ($\varepsilon_e, \varepsilon_n$) exponents
- **spacing** (Real) – spatial increment of the *level set grid*, if you picked a pre-defined *shape* or a *clump*
- **distField** (*list*) – the *discrete distance field* on *grid* (if given) as a list (of list of list; use .tolist() if working initially with 3D numpy arrays), where distField[i][j][k] is the distance value at grid.gridPoint(i,j,k)
- **grid** (*RegularGrid*) – the *grid carrying the distance field*, when the latter is directly assigned through *distField*
- **smearCoeff** (Real) – passed to *LevelSet.smearCoeff*
- **nSurfNodes** (*int*) – number of requested *surface nodes* when ray tracing them (number of rays, actually), passed to the corresponding argument of *LevelSet.rayTraceSurfNodes* together with *nodesPath* and *nodesTol* (exclusive of *surfNodes*)
- **nodesPath** (*int*) – path for ray tracing the *surface nodes*, passed to the corresponding argument of *LevelSet.rayTraceSurfNodes* (has to be used exclusive of *surfNodes*)
- **surfNodes** (*list*) – *surface nodes* as a list of Vector3r for a direct assignment of those, instead of ray tracing them while using *nSurfNodes* and *nodesPath* (a non-empty *surfNodes* is actually enough to bypass ray tracing and those other attributes and trigger direct assignment)
- **nodesTol** (Real) – tolerance while ray tracing the *surface nodes* (and not assigning them with *surfNodes*), passed to to the corresponding argument of *LevelSet.rayTraceSurfNodes*
- **n_neighborsNodes** (*int*) – passed to *LevelSet.n_neighborsNodes* for an optimized contact search in *Ig2_LevelSet_LevelSet_LSnodeGeom*

- **orientation** (Quaternion) – the initial orientation of the body
- **hasAABE** (*bool*) – flag indicating if the axis-aligned bounding ellipsoid (AABE) was set, passed to *LevelSet.hasAABE*
- **axesAABE** (Vector3) – principal half-axes of the axis aligned bounding ellipsoid (AABE) when *hasAABE*, passed to *LevelSet.axesAABE*
- **dynamic** (*bool*) – passed to *Body.dynamic*
- **material** (*Material*) – passed to *Body.material*
- **starLike** (*bool*) – passed to *LevelSet.starLike* when the function is also passed *grid* and *distField* (otherwise, *LevelSet.starLike* is automatically set)
- **color** – *rendering color* choice

Returns
a corresponding body instance

yade.utils.**loadVars**(*mark=None*)

Load variables from *utils.saveVars*, which are saved inside the simulation. If `mark==None`, all save variables are loaded. Otherwise only those with the mark passed.

yade.utils.**makeVideo**(*frameSpec*, *out*, *renameNotOverwrite=True*, *fps=24*, *kbps=6000*, *bps=None*)

Create a video from external image files using mencoder. Two-pass encoding using the default mencoder codec (mpeg4) is performed, running multi-threaded with number of threads equal to number of OpenMP threads allocated for Yade.

Parameters

- **frameSpec** – wildcard | sequence of filenames. If list or tuple, filenames to be encoded in given order; otherwise wildcard understood by mencoder's mf:// URI option (shell wildcards such as `/tmp/snap-*.png` or and printf-style pattern like `/tmp/snap-%05d.png`)
- **out** (*str*) – file to save video into
- **renameNotOverwrite** (*bool*) – if True, existing same-named video file will have *-number* appended; will be overwritten otherwise.
- **fps** (*int*) – Frames per second (`-mf fps=…`)
- **kbps** (*int*) – Bitrate (`-lavcopts vbitrate=…`) in kb/s

yade.utils.**perpendicularArea**(*axis*)

Return area perpendicular to given axis (0=x,1=y,2=z) generated by bodies for which the function consider returns True (defaults to returning True always) and which is of the type *Sphere*.

yade.utils.**phiIniPy**(*ioPyFn*, *grid*)

Returns a 3D discrete field appropriate to serve as *FastMarchingMethod.phiIni* (LS_DEM feature required), applying a user-made Python function *ioPyFn*

Parameters

- **ioPyFn** – an existing inside-outside Python function that takes three numbers (cartesian coordinates) as arguments
- **grid** (*RegularGrid*) – the *RegularGrid* instance to operate on

Return list
an appropriate 3D discrete field to pass at *FastMarchingMethod.phiIni*

yade.utils.**plotDirections**(*aabb=()*, *mask=0*, *bins=20*, *numHist=True*, *noShow=False*, *sphSph=False*)

Plot 3 histograms for distribution of interaction directions, in yz,xz and xy planes and (optional but default) histogram of number of interactions per body. If sphSph only sphere-sphere interactions are considered for the 3 directions histograms.

Returns

If *noShow* is `False`, displays the figure and returns nothing. If *noShow*, the figure object is returned without being displayed (works the same way as *plot.plot*).

`yade.utils.plotNumInteractionsHistogram(`*cutoff=0.0*`)`

Plot histogram with number of interactions per body, optionally cutting away *cutoff* relative axis-aligned box from specimen margin.

`yade.utils.polyhedron(`*vertices*, *fixed=False*, *wire=True*, *color=None*, *highlight=False*, *noBound=False*, *material=-1*, *mask=1*`)`

Create polyhedron with given parameters.

Parameters

vertices ([Vector3]) – coordinates of vertices in the global coordinate system.

See *utils.sphere*'s documentation for meaning of other parameters.

`yade.utils.psd(`*bins=5*, *mass=True*, *mask=-1*`)`

Calculates particle size distribution.

Parameters

- **bins** (*int*) – number of bins
- **mass** (*bool*) – if true, the mass-PSD will be calculated
- **mask** (*int*) – *Body.mask* for the body

Returns

- binsSizes: list of bin's sizes
- binsProc: how much material (in percents) are in the bin, cumulative
- binsSumCum: how much material (in units) are in the bin, cumulative

binsSizes, binsProc, binsSumCum

`yade.utils.randomColor(`*seed=None*`)`

Return random color from current style

`yade.utils.randomOrientation()`

Returns (uniformly distributed) random orientation. Taken from Eigen::Quaternion::UnitRandom() source code. Uses standard Python random.random() function(s), you can `random.seed()` it

`yade.utils.randomizeColors(`*onlyDynamic=False*`)`

Assign random colors to *Shape::color*.

If onlyDynamic is true, only dynamic bodies will have the color changed.

`yade.utils.readParamsFromTable(`*tableFileLine=None*, *noTableOk=True*, *unknownOk=False*, ***kw*`)`

Read parameters from a file and assign them to ___builtin___ variables.

The format of the file is as follows (commens starting with # and empty lines allowed):

```
# commented lines allowed anywhere
name1 name2 ... # first non-blank line are column headings
                # empty line is OK, with or without comment
val1    val2    ... # 1st parameter set
val2    val2    ... # 2nd
...
```

Assigned tags (the `description` column is synthesized if absent,see *utils.TableParamReader*);

O.tags['description']=... # assigns the description column; might be synthesized O.tags['params']="name1=val1,name2=val2,..." # all explicitly assigned parameters O.tags['defaultParams']="unassignedName1=defaultValue1,..." # parameters that were left at their defaults O.tags['d.id']=O.tags['id']+"."+O.tags['description'] O.tags['id.d']=O.tags['description']+"."+O.tags['id']

All parameters (default as well as settable) are saved using *utils.saveVars*`('table')`.

Parameters

- **tableFileLine** – string attribute to define which line number (as seen in a text editor) from wich text file (with one value per blank-separated columns) to get the values from. A ':' should appear between the two informations, e.g. 'file.table:4' to read the 4th line from file.table file
- **noTableOk** (*bool*) – if False, raise exception if the file cannot be open; use default values otherwise
- **unknownOk** (*bool*) – do not raise exception if unknown column name is found in the file, and assign it as well

Returns

number of assigned parameters

yade.utils.**replaceCollider**(*colliderEngine*)

Replaces collider (Collider) engine with the engine supplied. Raises error if no collider is in engines.

yade.utils.**runningInBatch**()

Tell whether we are running inside the batch or separately.

yade.utils.**saveVars**(*mark=''*, *loadNow=True*, ***kw*)

Save passed variables into the simulation so that it can be recovered when the simulation is loaded again.

For example, variables *a*, *b* and *c* are defined. To save them, use:

```
>>> saveVars('something',a=1,b=2,c=3)
>>> from yade.params.something import *
>>> a,b,c
(1, 2, 3)
```

those variables will be save in the .xml file, when the simulation itself is saved. To recover those variables once the .xml is loaded again, use `loadVars('something')` and they will be defined in the yade.params.*mark* module. The *loadNow* parameter calls *utils.loadVars* after saving automatically. If 'something' already exists, given variables will be inserted.

yade.utils.**sphere**(*center*, *radius*, *dynamic=None*, *fixed=False*, *wire=False*, *color=None*, *highlight=False*, *material=-1*, *mask=1*)

Create sphere with given parameters; mass and inertia computed automatically.

Last assigned material is used by default (*material* = -1), and utils.defaultMaterial() will be used if no material is defined at all.

Parameters

- **center** (Vector3) – center
- **radius** (*float*) – radius
- **dynamic** (*float*) – deprecated, see "fixed"
- **fixed** (*float*) – generate the body with all DOFs blocked?
- **material** –

  specify *Body.material*; different types are accepted:

- int: O.materials[material] will be used; as a special case, if material==-1 and there is no shared materials defined, utils.defaultMaterial() will be assigned to O.materials[0]
- string: label of an existing material that will be used
- *Material* instance: this instance will be used
- callable: will be called without arguments; returned Material value will be used (Material factory object, if you like)

- **mask** (*int*) – *Body.mask* for the body
- **wire** – display as wire sphere?
- **color** – *rendering color* choice. A random color is assigned in the absence of an explicit user choice
- **highlight** – highlight this body in the viewer?
- **Vector3-or-None** – body's color, as normalized RGB; random color will be assigned if `None`.

Returns

A Body instance with desired characteristics.

Creating default shared material if none exists neither is given:

```
>>> O.reset()
>>> from yade import utils
>>> len(O.materials)
0
>>> s0=utils.sphere([2,0,0],1)
>>> len(O.materials)
1
```

Instance of material can be given:

```
>>> s1=utils.sphere([0,0,0],1,wire=False,color=(0,1,
↪0),material=ElastMat(young=30e9,density=2e3))
>>> s1.shape.wire
False
>>> s1.shape.color
Vector3(0,1,0)
>>> s1.mat.density
2000.0
```

Material can be given by label:

```
>>> O.materials.append(FrictMat(young=10e9,poisson=.11,label='myMaterial'))
1
>>> s2=utils.sphere([0,0,2],1,material='myMaterial')
>>> s2.mat.label
'myMaterial'
>>> s2.mat.poisson
0.11
```

Finally, material can be a callable object (taking no arguments), which returns a Material instance. Use this if you don't call this function directly (for instance, through yade.pack.randomDensePack), passing only 1 *material* parameter, but you don't want material to be shared.

For instance, randomized material properties can be created like this:

```
>>> import random
>>> def matFactory(): return ElastMat(young=1e10*random.
↪random(),density=1e3+1e3*random.random())
...
>>> s3=utils.sphere([0,2,0],1,material=matFactory)
>>> s4=utils.sphere([1,2,0],1,material=matFactory)
```

yade.utils.**tetra**(*vertices*, *strictCheck=True*, *fixed=False*, *wire=True*, *color=None*, *highlight=False*, *noBound=False*, *material=-1*, *mask=1*)

Create tetrahedron with given parameters.

Parameters

- **vertices** (*[Vector3,Vector3,Vector3,Vector3]*) – coordinates of vertices in the global coordinate system.
- **strictCheck** (*bool*) – checks vertices order, raise RuntimeError for negative volume

See *utils.sphere*'s documentation for meaning of other parameters.

yade.utils.**tetraPoly**(*vertices*, *fixed=False*, *wire=True*, *color=None*, *highlight=False*, *noBound=False*, *material=-1*, *mask=1*)

Create tetrahedron (actually simple Polyhedra) with given parameters.

Parameters

vertices (*[Vector3,Vector3,Vector3,Vector3]*) – coordinates of vertices in the global coordinate system.

See *utils.sphere*'s documentation for meaning of other parameters.

yade.utils.**trackPerfomance**(*updateTime=5*)

Track perfomance of a simulation. (Experimental) Will create new thread to produce some plots. Useful for track perfomance of long run simulations (in bath mode for example).

yade.utils.**typedEngine**(*name*)

Return first engine from current O.engines, identified by its type (as string). For example:

```
>>> from yade import utils
>>> O.engines=[InsertionSortCollider(),NewtonIntegrator(),GravityEngine()]
>>> utils.typedEngine("NewtonIntegrator") == O.engines[1]
True
```

yade.utils.**uniaxialTestFeatures**(*filename=None*, *areaSections=10*, *axis=-1*, *distFactor=2.2*, ***kw*)

Get some data about the current packing useful for uniaxial test:

1. Find the dimensions that is the longest (uniaxial loading axis)
2. Find the minimum cross-section area of the specimen by examining several (areaSections) sections perpendicular to axis, computing area of the convex hull for each one. This will work also for non-prismatic specimen.
3. Find the bodies that are on the negative/positive boundary, to which the straining condition should be applied.

Parameters

- **filename** – if given, spheres will be loaded from this file (ASCII format); if not, current simulation will be used.
- **areaSection** (*float*) – number of section that will be used to estimate cross-section

- **axis** (*{0,1,2}*) – if given, force strained axis, rather than computing it from predominant length

Returns
dictionary with keys `negIds`, `posIds`, `axis`, `area`.

Warning

The function *utils.approxSectionArea* uses convex hull algorithm to find the area, but the implementation is reported to be *buggy* (bot works in some cases). Always check this number, or fix the convex hull algorithm (it is documented in the source, see py/_utils.cpp).

`yade.utils.vmData()`

Return memory usage data from Linux's /proc/[pid]/status, line VmData.

`yade.utils.voxelPorosityTriaxial(`*triax*, *resolution=200*, *offset=0*`)`

Calculate the porosity of a sample, given the TriaxialCompressionEngine.

A function *utils.voxelPorosity* is invoked, with the volume of a box enclosed by TriaxialCompressionEngine walls. The additional parameter offset allows using a smaller volume inside the box, where each side of the volume is at offset distance from the walls. By this way it is possible to find a more precise porosity of the sample, since at walls' contact the porosity is usually reduced.

A recommended value of offset is bigger or equal to the average radius of spheres inside.

The value of resolution depends on size of spheres used. It can be calibrated by invoking voxelPorosityTriaxial with offset=0 and comparing the result with TriaxialCompressionEngine.porosity. After calibration, the offset can be set to radius, or a bigger value, to get the result.

Parameters

- **triax** – the TriaxialCompressionEngine handle
- **resolution** – voxel grid resolution
- **offset** – offset distance

Returns
the porosity of the sample inside given volume

Example invocation:

```
from yade import utils
rAvg=0.03
TriaxialTest(numberOfGrains=200,radiusMean=rAvg).load()
O.dt=-1
O.run(1000)
O.engines[4].porosity
0.44007807740143889
utils.voxelPorosityTriaxial(O.engines[4],200,0)
0.44055412500000002
utils.voxelPorosityTriaxial(O.engines[4],200,rAvg)
0.36798199999999998
```

`yade.utils.waitIfBatch()`

Block the simulation if running inside a batch. Typically used at the end of script so that it does not finish prematurely in batch mode (the execution would be ended in such a case).

`yade.utils.wall(`*position*, *axis*, *sense=0*, *color=Vector3(0.80000000000000444, 0.80000000000000444, 0.59999999999999778)*, *material=-1*, *mask=1*`)`

Return ready-made wall body.

Parameters

- **position** (*float-or-Vector3*) – center of the wall. If float, it is the position along given axis, the other 2 components being zero
- **axis** (*{0,1,2}*) – orientation of the wall normal (0,1,2) for x,y,z (sc. planes yz, xz, xy)
- **sense** (*{-1,0,1}*) – sense in which to interact (0: both, -1: negative, +1: positive; see *Wall*)

See *utils.sphere*'s documentation for meaning of other parameters.

yade.utils.**xMirror**(*half*)

Mirror a sequence of 2d points around the x axis (changing sign on the y coord). The sequence should start up and then it will wrap from y downwards (or vice versa). If the last point's x coord is zero, it will not be duplicated.

yade._utils.**PWaveTimeStep**() → float

Get timestep accoring to the velocity of P-Wave propagation; computed for spheres and/or polyhedra based on their sizes, rigidities and masses.

yade._utils.**RayleighWaveTimeStep**() → float

Determination of time step according to Rayleigh wave speed of force propagation.

yade._utils.**TetrahedronCentralInertiaTensor**(*(object)arg1*) → *yade._minieigenHP.Matrix3*

TODO

yade._utils.**TetrahedronInertiaTensor**(*(object)arg1*) → *yade._minieigenHP.Matrix3*

TODO

yade._utils.**TetrahedronSignedVolume**(*(object)arg1*) → float

TODO

yade._utils.**TetrahedronVolume**(*(object)arg1*) → float

TODO

yade._utils.**TetrahedronWithLocalAxesPrincipal**(*(yade.wrapper.Body)arg1*) → *yade._minieigenHP.Quaternion*

TODO

yade._utils.**aabbExtrema**(*[(float)cutoff=0.0[, (bool)centers=False]]*) → tuple

Return coordinates of box enclosing all spherical bodies

Parameters

- **centers** (*bool*) – do not take sphere radii in account, only their centroids
- **cutoff** (*float (0…1)*) – relative dimension by which the box will be cut away at its boundaries.

Returns

[lower corner, upper corner] as [Vector3,Vector3]

yade._utils.**angularMomentum**(*[(yade._minieigenHP.Vector3)origin=Vector3(0, 0, 0)]*) → *yade._minieigenHP.Vector3*

Returns total angular momentum of the simulation, about input point *origin*

yade._utils.**approxSectionArea**(*(float)arg1*, *(int)arg2*) → float

Compute area of convex hull when when taking (swept) spheres crossing the plane at coord, perpendicular to axis.

yade._utils.**bodyNumInteractionsHistogram**(*(tuple)aabb*) → tuple

yade._utils.bodyStressTensors() → list

Compute and return a table with per-particle stress tensors. Each tensor represents the average stress in one particle, obtained from the contour integral of applied load as detailed below. This definition is considering each sphere as a continuum. It can be considered exact in the context of spheres at static equilibrium, interacting at contact points with negligible volume changes of the solid phase (this last assumption is not restricting possible deformations and volume changes at the packing scale).

Proof:

First, we remark the identity: $\sigma_{ij} = \delta_{ik}\sigma_{kj} = x_{i,k}\sigma_{kj} = (x_i\sigma_{kj})_{,k} - x_i\sigma_{kj,k}$.

At equilibrium, the divergence of stress is null: $\sigma_{kj,k} = 0$. Consequently, after divergence theorem: $\frac{1}{V}\int_V \sigma_{ij}\,dV = \frac{1}{V}\int_V (x_i\sigma_{kj})_{,k}\,dV = \frac{1}{V}\int_{\partial V} x_i\sigma_{kj}n_k\,dS = \frac{1}{V}\sum_b x_i^b f_j^b$.

The last equality is implicitely based on the representation of external loads as Dirac distributions whose zeros are the so-called *contact points*: 0-sized surfaces on which the *contact forces* are applied, located at x_i in the deformed configuration.

A weighted average of per-body stresses will give the average stress inside the solid phase. There is a simple relation between the stress inside the solid phase and the stress in an equivalent continuum in the absence of fluid pressure. For porosity n, the relation reads: $\sigma_{ij}^{equ.} = (1-n)\sigma_{ij}^{solid}$.

This last relation may not be very useful if porosity is not homogeneous. If it happens, one can define the equivalent bulk stress a the particles scale by assigning a volume to each particle. This volume can be obtained from *TesselationWrapper* (see e.g. [Catalano2014a])

yade._utils.calm([*(int)mask=-1*]) → None

Set translational and rotational velocities of bodies to zero. Applied to all *dynamic* bodies by default. To calm only some of them, use mask parameter, it will calm only dynamic bodies with groupMask compatible to given value

yade._utils.cart2spher(*(yade._minieigenHP.Vector3)vec*) → *yade._minieigenHP.Vector3*

Converts cartesian coordinates to spherical ones.

Parameters

vec (Vector3) – the (x, y, z) cartesian coordinates

Returns

a (r, ϑ, φ) Vector3 of spherical coordinates, with $\vartheta = (\boldsymbol{e_z}, \boldsymbol{e_r}) \in [0; \pi]$ and $\varphi \in [0; 2\pi]$ measured in (x, y) plane from x-axis

yade._utils.coordsAndDisplacements(*(int)axis*[, *(tuple)Aabb=()*]) → tuple

Return tuple of 2 same-length lists for coordinates and displacements (coordinate minus reference coordinate) along given axis (1st arg); if the Aabb=((x_min,y_min,z_min),(x_max,y_max,z_max)) box is given, only bodies within this box will be considered.

yade._utils.createInteraction(*(int)id1*, *(int)id2*[, *(bool)virtualI=False*]) → yade.wrapper.Interaction

Create interaction between given bodies by hand.

If *virtualI=False*, current engines are searched for *IGeomDispatcher* and *IPhysDispatcher* (might be both hidden in *InteractionLoop*). Geometry is created using **force** parameter of the *geometry dispatcher*, wherefore the interaction will exist even if bodies do not spatially overlap and the functor would return **false** under normal circumstances.

If *virtualI=True* the interaction is left in a virtual state.

Warning

This function will very likely behave incorrectly for periodic simulations (though it could be extended it to handle it farily easily).

yade._utils.**distApproxRose**(*(yade._minieigenHP.Vector3)pt*) → float

An approximate distance value to a rose-like flaky surface defined in spherical coordinates as $r = 3 + 1.5$ sin(5 theta) sin(4 phi) (see *cart2spher* function for spherical <→ cartesian convention).

Parameters
pt (Vector3) – the pt of interest given in (x, y, z) cartesian coordinates.

Returns
a 0-approximation distance value.

yade._utils.**distApproxSE**(*(yade._minieigenHP.Vector3)pt*, *(yade._minieigenHP.Vector3)radii*, *(yade._minieigenHP.Vector2)epsilons*) → float

An approximate distance value to a superellipsoid surface defined (in local axes) as $f(x, y, z) = (|x/r_x|^{2/\varepsilon_e} + |y/r_y|^{2/\varepsilon_e})^{\varepsilon_e/\varepsilon_n} + |z/r_z|^{2/\varepsilon_n} = 1$.

Parameters

- **pt** (Vector3) – the (x, y, z) of interest
- **radii** (Vector3) – the (r_x, r_y, r_z)
- **epsilons** (Vector2) – the $(\varepsilon_e, \varepsilon_n)$ exponents

Returns
a 0-approximation distance value.

yade._utils.**distIniClump**(*(yade.wrapper.Clump)clump*, *(yade.wrapper.RegularGrid)grid*) → object

An appropriate discrete field to serve as a Fast Marching Method input in *FastMarchingMethod.phiIni* in order to compute distance to a clump.

Parameters

- **clump** (*Clump*) – considered clump instance
- **grid** (*RegularGrid*) – the *RegularGrid* instance to consider

Returns
an appropriate 3D discrete field for *FastMarchingMethod.phiIni*.

yade._utils.**distIniSE**(*(yade._minieigenHP.Vector3)radii*, *(yade._minieigenHP.Vector2)epsilons*, *(yade.wrapper.RegularGrid)grid*) → object

An appropriate discrete field to serve as a Fast Marching Method input in *FastMarchingMethod.phiIni* in order to compute distance to a superellipsoid.

Parameters

- **radii** (Vector3) – the (r_x, r_y, r_z)
- **epsilons** (Vector2) – the $(\varepsilon_e, \varepsilon_n)$ exponents
- **grid** (*RegularGrid*) – the *RegularGrid* instance to consider

Returns
an appropriate 3D discrete field for *FastMarchingMethod.phiIni*.

yade._utils.**fabricTensor**(*[(float)cutoff=0.0[, (bool)splitTensor=False[, (float)thresholdForce=nan[, (object)extrema=[]]]]]*) → tuple

Computes the fabric tensor $F_{ij} = \frac{1}{n_c} \sum_c n_i n_j$ [Satake1982] over all interactions c with n_c the total number of interactions and n_i and n_j the i and j components of the contact normal, respectively.

Parameters

- **cutoff** (Real) – intended to disregard boundary effects: to define in [0;1] to focus on the interactions located in the centered inner (1-cutoff)^3*V part of the spherical packing V.
- **splitTensor** (*bool*) – split the fabric tensor into two parts related to the strong (greatest compressive normal forces) and weak contact forces respectively.
- **thresholdForce** (Real) – if the fabric tensor is split into two parts, a threshold value can be specified otherwise the mean contact force is considered by default. Use negative signed values for compressive states. To note that this value could be set to zero if one wanted to make distinction between compressive and tensile forces.
- **extrema** (*list*) – defines through a two-Vector3-list (min,max) an axis aligned box that restricts the interactions to consider. A value has to be given for the function to be effective with non-spherical particles.

yade._utils.flipCell() → *yade._minieigenHP.Matrix3*

utils.flipCell is deprecated, use *O.cell.flipCell* or *O.cell.flipFlippable*

yade._utils.forcesOnCoordPlane(*(float)arg1*, *(int)arg2*) → *yade._minieigenHP.Vector3*

yade._utils.forcesOnPlane(*(yade._minieigenHP.Vector3)planePt*, *(yade._minieigenHP.Vector3)normal*) → *yade._minieigenHP.Vector3*

Find all interactions deriving from *NormShearPhys* that cross given plane and sum forces (both normal and shear) on them.

Parameters

- **planePt** (Vector3) – a point on the plane
- **normal** (Vector3) – plane normal (will be normalized).

yade._utils.getBodyIdsContacts([*(int)bodyID=0*]) → list

Get a list of body-ids, which contacts the given body.

yade._utils.getCapillaryStress([*(float)volume=0*[, *(bool)mindlin=False*]]) → *yade._minieigenHP.Matrix3*

Compute and return Love-Weber capillary stress tensor:

$\sigma_{ij}^{cap} = \frac{1}{V}\sum_b l_i^b f_j^{cap,b}$, where the sum is over all interactions, with l the branch vector (joining centers of the bodies) and f^{cap} is the capillary force. V can be passed to the function. If it is not, it will be equal to one in non-periodic cases, or equal to the volume of the cell in periodic cases. Only the CapillaryPhys interaction type is supported presently. Using this function with physics MindlinCapillaryPhys needs to pass True as second argument.

yade._utils.getDepthProfiles(*(float)volume*, *(int)nCell*, *(float)dz*, *(float)zRef*[, *(bool)activateCond=False*[, *(float)radiusPy=0*[, *(int)direction=2*]]]) → tuple

Compute and return the particle velocity and solid volume fraction (porosity) depth profile along the direction specified (default is z; 0=>x,1=>y,2=>z). For each defined cell z, the k component of the average particle velocity reads:

$< v_k >^z = \sum_p V^p v_k^p / \sum_p V^p$,

where the sum is made over the particles contained in the cell, v_k^p is the k component of the velocity associated to particle p, and V^p is the part of the volume of the particle p contained inside the cell. This definition allows to smooth the averaging, and is equivalent to taking into account the center of the particles only when there is a lot of particles in each cell. As for the solid volume fraction, it is evaluated in the same way: for each defined cell z, it reads:

$< \varphi >^z = \frac{1}{V_{cell}} \sum_p V^p$, **where V_{cell} is the volume of the cell considered, and V^p is the volume of particle p contained in cell z.**

This function gives depth profiles of average velocity and solid volume fraction, returning the average quantities in each cell of height dz, from the reference horizontal plane at elevation zRef (input parameter) until the plane of elevation zRef+nCell*dz (input parameters). If the argument activateCond is set to true, do the average only on particles of radius equal to radiusPy (input parameter)

yade._utils.**getDepthProfiles_center**(*(float)volume*, *(int)nCell*, *(float)dz*, *(float)zRef*[, *(bool)activateCond=False*[, *(float)radiusPy=0*]]) → tuple

Same as getDepthProfiles but taking into account particles as points located at the particle center.

yade._utils.**getDynamicStress**() → list

Compute the dynamic stress tensor for each body: $\sigma_D^p = -\frac{1}{V^p} m^p u'^p \otimes u'^p$

yade._utils.**getSlicedProfiles**(*(float)vCell*, *(int)nCell*, *(float)dP*, *(object)sliceCenters*, *(object)sliceWidths*, *(float)refP*, *(float)refS*[, *(int)dirP=2*[, *(int)dirS=1*[, *(bool)activateCond=False*[, *(float)radiusPy=0*[, *(float)nSimpson=50*]]]]]) → tuple

Compute and return the particle solid volume fraction (porosity) and velocity profiles along a specific direction dirP and for a given subdomain. In the direction dirP, the subdomain is divided into nCell of size dP. For each cell, the averaged solid volume fraction reads:

$< \varphi >= \frac{1}{V_{cell}} \sum_p V^p$

and the averaged particle velocity reads:

$< v_k >= \sum_p V^p v_k^p / \sum_p V^p$,

where the sum is made over the particles p contained in the cell, v_k^p is the k component of the velocity associated to particle p, V^p is the part of the volume of the particle p contained inside the cell, and V_{cell} is the volume of the cell (all subdomain slices combined). The volume of the sliced particle V^p is computed analytically when the particle is not sliced by the subdomain boundaries. Otherwise, V^p is computed using a Simpson integration of the sliced area of the sliced sphere.

This function allows to define a discontinuous subdomain made of different slices in direction dirS. This can be useful to exclude specific zones from the averaging procedure or to target similar zones like symmetric boundaries.

Arguments are: vCell : volume of a cell, all slices combined. (e.g. dP*length*(slicewidth1+slicewidth2)) nCell : number of cells in the profile direction dP : discretisation interval in the Profile direction, sliceCenters : array containing the position of the center of each slice from refS in the S direction, sliceWidths : array containing the width of each slice, refP : reference position in the Profile direction, refS : reference position in the slice direction, dirP : direction of the profile (0:x, 1:y, 2:z), (default:2), dirS : direction of the slices (0:x, 1:y, 2:z), must be different from dirP, (default:1), activateCond : if true, will only consider particle of radius equal radiusPy, (default:false), nSimpson : number of intervals per particle radius for the Simpson integration, (default:50),

yade._utils.**getSpheresMass**([*(int)mask=-1*]) → float

Compute the total mass of spheres in the simulation, mask parameter is considered

yade._utils.**getSpheresVolume**([*(int)mask=-1*]) → float

Compute the total volume of spheres in the simulation, mask parameter is considered

yade._utils.**getSpheresVolume2D**([*(int)mask=-1*]) → float

Compute the total volume of discs in the simulation, mask parameter is considered

yade._utils.**getStress**([*(float)volume=0*]) → *yade._minieigenHP.Matrix3*

Compute and return Love-Weber stress tensor:

$\sigma_{ij} = \frac{1}{V}\sum_b f_i^b l_j^b$, where the sum is over all interactions, with f the contact force and l the branch vector (joining centers of the bodies). Stress is negativ for repulsive contact forces, i.e. compression. V can be passed to the function. If it is not, it will be equal to the volume of the cell in periodic cases, or to the one deduced from utils.aabbDim() in non-periodic cases.

yade._utils.**getStressAndTangent**(*[(float)volume=0[, (bool)symmetry=True]]*) → tuple

Compute overall stress of periodic cell using the same equation as function getStress. In addition, the tangent operator is calculated using the equation published in [Kruyt and Rothenburg1998]_:

$S_{ijkl} = \frac{1}{V}\sum_c (k_n n_i l_j n_k l_l + k_t t_i l_j t_k l_l)$

Although the above formula gives the elements of the fourth-order stiffness tensor, this tangent operator will be returned in Voigt notation, giving a 6 by 6 tensor, where elements of S_{ijkl} mapping to the same Voigt element will be averaged.

Parameters

- **volume** (*float*) – same as in function getStress
- **symmetry** (*bool*) – make the tensors symmetric.

Returns

macroscopic stress tensor and tangent operator as py::tuple

yade._utils.**getStressProfile**(*(float)volume*, *(int)nCell*, *(float)dz*, *(float)zRef*, *(object)vPartAverageX*, *(object)vPartAverageY*, *(object)vPartAverageZ*) → tuple

Compute and return the stress tensor depth profile, including the contribution from Love-Weber stress tensor and the dynamic stress tensor taking into account the effect of particles inertia. For each defined cell z, the stress tensor reads:

$\sigma_{ij}^z = \frac{1}{V}\sum_c f_i^c l_j^{c,z} - \frac{1}{V}\sum_p m^p u_i'^p u_j'^p$,

where the first sum is made over the contacts which are contained or cross the cell z, f^c is the contact force from particle 1 to particle 2, and l^{c,z} is the part of the branch vector from particle 2 to particle 1, contained in the cell. The second sum is made over the particles, and u'^p is the velocity fluctuations of the particle p with respect to the spatial averaged particle velocity at this point (given as input parameters). The expression of the stress tensor is the same as the one given in getStress plus the inertial contribution. Apart from that, the main difference with getStress stands in the fact that it gives a depth profile of stress tensor, i.e. from the reference horizontal plane at elevation zRef (input parameter) until the plane of elevation zRef+nCell*dz (input parameters), it is computing the stress tensor for each cell of height dz. For the love-Weber stress contribution, the branch vector taken into account in the calculations is only the part of the branch vector contained in the cell considered. To validate the formulation, it has been checked that activating only the Love-Weber stress tensor, and suming all the contributions at the different altitude, we recover the same stress tensor as when using getStress. For my own use, I have troubles with strong overlap between fixed object, so that I made a condition to exclude the contribution to the stress tensor of the fixed objects, this can be desactivated easily if needed (and should be desactivated for the comparison with getStress).

yade._utils.**getStressProfile_contact**(*(float)volume*, *(int)nCell*, *(float)dz*, *(float)zRef*) → tuple

same as getStressProfile, only contact contribution.

yade._utils.**getTotalDynamicStress**(*[(float)volume=0]*) → *yade._minieigenHP.Matrix3*

Compute the total dynamic stress tensor : $\sigma_D = -\frac{1}{V}\sum_p m^p u'^p \otimes u'^p$. The volume have to be provided for non-periodic simulations. It is computed from cell volume for periodic simulations.

yade._utils.**getViscoelasticFromSpheresInteraction**(*(float)tc*, *(float)en*, *(float)es*) → dict

Attention! The function is deprecated! Compute viscoelastic interaction parameters from analytical

solution of a pair spheres collision problem:

$$k_n = \frac{m}{t_c^2}\left(\pi^2 + (\ln e_n)^2\right)$$
$$c_n = -\frac{2m}{t_c}\ln e_n$$
$$k_t = \frac{2}{7}\frac{m}{t_c^2}\left(\pi^2 + (\ln e_t)^2\right)$$
$$c_t = -\frac{2}{7}\frac{m}{t_c}\ln e_t$$

where k_n, c_n are normal elastic and viscous coefficients and k_t, c_t shear elastic and viscous coefficients. For details see [Pournin2001].

Parameters

- **m** (*float*) – sphere mass m
- **tc** (*float*) – collision time t_c
- **en** (*float*) – normal restitution coefficient e_n
- **es** (*float*) – tangential restitution coefficient e_s

Returns

dictionary with keys **kn** (the value of k_n), **cn** (c_n), **kt** (k_t), **ct** (c_t).

yade._utils.**growParticle**(*(int)bodyID*, *(float)multiplier*[, *(bool)updateMass=True*]) → None

Change the size of a single sphere (to be implemented: single clump). If updateMass=True, then the mass is updated.

yade._utils.**growParticles**(*(float)multiplier*[, *(bool)updateMass=True*[, *(bool)dynamicOnly=True*]]) → None

Change the size of spheres and clumps of spheres by the multiplier. If updateMass=True, then the mass and inertia are updated. dynamicOnly=True will select dynamic bodies.

yade._utils.**highlightNone**() → None

Reset *highlight* on all bodies.

yade._utils.**initMPI**() → None

Initialize MPI communicator, for Foam Coupling

yade._utils.**inscribedCircleCenter**(*(yade._minieigenHP.Vector3)v1*, *(yade._minieigenHP.Vector3)v2*, *(yade._minieigenHP.Vector3)v3*) → *yade._minieigenHP.Vector3*

Return center of inscribed circle for triangle given by its vertices *v1*, *v2*, *v3*.

yade._utils.**insideClump**(*(yade._minieigenHP.Vector3)pt*, *(yade.wrapper.Clump)clump*) → bool

Tells whether some point is inside or outside a clump

Parameters

- **pt** (Vector3) – the point of interest expressed in local coordinates
- **clump** (*Clump*) – the clump instance to consider

Return bool

True when strictly inside, False otherwise

yade._utils.**interactionAnglesHistogram**(*(int)axis*[, *(int)mask=0*[, *(int)bins=20*[, *(tuple)aabb=()*[, *(bool)sphSph=0*[, *(float)minProjLen=1e-06*]]]]]) → tuple

yade._utils.**intrsOfEachBody**() → list

returns list of lists of interactions of each body

yade._utils.**kineticEnergy**([*(bool)findMaxId=False*]) → object

Compute overall kinetic energy of the simulation as

$$\sum \frac{1}{2}\left(m_i \boldsymbol{v}_i^2 + \boldsymbol{\omega}(\mathbf{I}\boldsymbol{\omega}^T)\right).$$

For *asphерical* bodies, necessary frame transformations are applied to the inertia tensor $\mathbf{I}$ as stored in *state.inertia*.

yade._utils.**lsSimpleShape**(*(int)shape*, *(yade._minieigenHP.AlignedBox3)aabb*[, *(float)step=0.1*[, *(float)smearCoeff=1.5*[, *(yade._minieigenHP.Vector2)epsilons=Vector2(0, 0)*[, *(yade.wrapper.Clump)clump=<Clump instance at 0x212a1c0>*]]]]) → yade.wrapper.LevelSet

Creates a LevelSet shape among pre-defined ones. Not intended to be used directly, see levelSetBody() instead.

Parameters

- **shape** (*int*) – a shape index among supported choices
- **aabb** (AlignedBox3) – the axis-aligned surrounding box of the body
- **step** (Real) – the LevelSet grid step size
- **smearCoeff** (Real) – passed to LevelSet.smearCoeff
- **epsilons** (Vector2) – the epsilon exponents in case *shape* = 3 (superellipsoid)
- **clump** (*Clump*) – the Clump instance to mimick in case *shape* = 4

Returns

a LevelSet instance.

yade._utils.**maxOverlapRatio**() → float

Return maximum overlap ration in interactions (with *ScGeom*) of two *spheres*. The ratio is computed as $\frac{u_N}{2(r_1 r_2)/r_1+r_2}$, where u_N is the current overlap distance and r_1, r_2 are radii of the two spheres in contact.

yade._utils.**momentum**() → *yade._minieigenHP.Vector3*

Returns total linear momentum of the simulation

yade._utils.**nGP**(*(float)min*, *(float)max*, *(float)step*) → int

Defines how many gridpoints are necessary to go from *min* to (at least) *max*, by *step* increments, eg when constructing a *RegularGrid*

param Real min
lowest grid extremity as (min,min,min) in case you just give a number, or as (min[0],min[1],min[2]) in case you give a tuple/list/Vector3

param Real max
highest gridpoint as (max,max,max) in case you give a number, or as (max[0],max[1],max[2]) in case you give a tuple/list/Vector3. The actual highest point of the grid may be beyond max by something like *step*.

param Real step
the distance between two consecutive grid points (the same along each axis).

return
either an integer, or a Vector3 of, depending on usage

nGP((yade._minieigenHP.Vector3)min, (yade._minieigenHP.Vector3)max, (float)step) → yade._minieigenHP.Vector3i :
Type-overloaded version of the above, to allow for both types of max/min attributes.

yade._utils.**negPosExtremeIds**(*(int)axis*, *(float)distFactor*) → tuple

Return list of ids for spheres (only) that are on extremal ends of the specimen along given axis; distFactor multiplies their radius so that sphere that do not touch the boundary coordinate can also be returned.

yade._utils.**normalShearStressTensors**([*(bool)compressionPositive=False*[, *(bool)splitNormalTensor=False*[, *(float)thresholdForce=nan*]]]) → tuple

Compute overall stress tensor of the periodic cell decomposed in 2 parts, one contributed by normal forces, the other by shear forces. The formulation can be found in [Thornton2000], eq. (3):

$$\sigma_{ij} = \frac{2}{V}\sum \mathsf{RN}\mathbf{n}_i\mathbf{n}_j + \frac{2}{V}\sum \mathsf{RT}\mathbf{n}_i\mathbf{t}_j$$

where V is the cell volume, R is "contact radius" (in our implementation, current distance between particle centroids), $\mathbf{n}$ is the normal vector, $\mathbf{t}$ is a vector perpendicular to $\mathbf{n}$, N and T are norms of normal and shear forces.

Parameters

- **splitNormalTensor** (*bool*) – if true the function returns normal stress tensor split into two parts according to the two subnetworks of strong an weak forces.
- **thresholdForce** (Real) – threshold value according to which the normal stress tensor can be split (e.g. a zero value would make distinction between tensile and compressive forces).

yade._utils.**numIntrsOfEachBody**() → list

returns list of number of interactions of each body

yade._utils.**phiIniCppPy**(*(yade.wrapper.RegularGrid)grid*) → object

A possibly handy function to construct a *FastMarchingMethod.phiIni* after applying on *grid* an inside-outside Python function. The latter necessarily names ioFn and takes three floating numbers (cartesian coordinates) as arguments. Code source of the present function is both C++ and Python, and execution should be faster and heavier in memory than the pure Python version utils.phiIniPy, both being under the second and few MB for grids with $\sim 10^4$ gridpoints.

Parameters
grid (*RegularGrid*) – the yref:*RegularGrid* instance to operate on with a preexisting ioFn Python function

Returns
an appropriate 3D discrete field for *FastMarchingMethod.phiIni*

yade._utils.**pointInsidePolygon**(*(tuple)arg1*, *(object)arg2*) → bool

yade._utils.**porosity**([*(float)volume=-1*]) → float

Compute packing porosity $\frac{V-V_s}{V}$ where V is overall volume and V_s is volume of spheres.

Parameters
volume (*float*) – overall volume V. For periodic simulations, current volume of the *Cell* is used. For aperiodic simulations, the value deduced from utils.aabbDim() is used. For compatibility reasons, positive values passed by the user are also accepted in this case.

yade._utils.**ptInAABB**(*(yade._minieigenHP.Vector3)arg1*, *(yade._minieigenHP.Vector3)arg2*, *(yade._minieigenHP.Vector3)arg3*) → bool

Return True/False whether the point p is within box given by its min and max corners

yade._utils.scalarOnColorScale(*(float)x*[, *(float)xmin=0*[, *(float)xmax=1*]]) → *yade._minieigenHP.Vector3*

Map scalar variable to color scale.

Parameters

- **x** (*float*) – scalar value which the function applies to.
- **xmin** (*float*) – minimum value for the color scale, with a return value of (0,0,1) for $x \leq xmin$, i.e. blue color in RGB.
- **xmax** (*float*) – maximum value, with a return value of (1,0,0) for $x \geq xmax$, i.e. red color in RGB.

Returns

a Vector3 depending on the relative position of x on a [*xmin*;*xmax*] scale.

yade._utils.setBodyAngularVelocity(*(int)id*, *(yade._minieigenHP.Vector3)angVel*) → None

Set a body angular velocity from its id and a new Vector3r.

Parameters

- **id** (*int*) – the body id.
- **angVel** (Vector3) – the desired updated angular velocity.

yade._utils.setBodyColor(*(int)id*, *(yade._minieigenHP.Vector3)color*) → None

Set a body color from its id and a new Vector3r.

Parameters

- **id** (*int*) – the body id.
- **color** (Vector3) – the desired updated color.

yade._utils.setBodyOrientation(*(int)id*, *(yade._minieigenHP.Quaternion)ori*) → None

Set a body orientation from its id and a new Quaternionr.

Parameters

- **id** (*int*) – the body id.
- **ori** (Quaternion) – the desired updated orientation.

yade._utils.setBodyPosition(*(int)id*, *(yade._minieigenHP.Vector3)pos*[, *(str)axis='xyz'*]) → None

Set a body position from its id and a new vector3r.

Parameters

- **id** (*int*) – the body id.
- **pos** (Vector3) – the desired updated position.
- **axis** (*str*) – the axis along which the position has to be updated (ex: if axis=="xy" and pos==Vector3r(r0,r1,r2), r2 will be ignored and the position along z will not be updated).

yade._utils.setBodyVelocity(*(int)id*, *(yade._minieigenHP.Vector3)vel*[, *(str)axis='xyz'*]) → None

Set a body velocity from its id and a new vector3r.

Parameters

- **id** (*int*) – the body id.
- **vel** (Vector3) – the desired updated velocity.
- **axis** (*str*) – the axis along which the velocity has to be updated (ex: if axis=="xy" and vel==Vector3r(r0,r1,r2), r2 will be ignored and the velocity along z will not be updated).

yade._utils.**setContactFriction**(*(float)angleRad*) → None

Modify the friction angle (in radians) inside the material classes and existing contacts. The friction for non-dynamic bodies is not modified.

yade._utils.**setNewVerticesOfFacet**(*(yade.wrapper.Body)b*, *(yade._minieigenHP.Vector3)v1*, *(yade._minieigenHP.Vector3)v2*, *(yade._minieigenHP.Vector3)v3*) → None

Sets new vertices (in global coordinates) to given facet.

yade._utils.**setRefSe3**() → None

Set reference *positions* and *orientations* of all *bodies* equal to their current *positions* and *orientations*.

yade._utils.**shiftBodies**(*(list)ids*, *(yade._minieigenHP.Vector3)shift*) → float

Shifts bodies listed in ids without updating their velocities.

yade._utils.**spher2cart**(*(yade._minieigenHP.Vector3)vec*) → *yade._minieigenHP.Vector3*

Converts spherical coordinates to cartesian ones.

Parameters

vec (Vector3) – the (r, ϑ, φ) spherical coordinates, see cart2spher function for conventions

Returns

a (x, y, z) Vector3 of cartesian coordinates

yade._utils.**spiralProject**(*(yade._minieigenHP.Vector3)pt*, *(float)dH_dTheta*[, *(int)axis=2*[, *(float)periodStart=nan*[, *(float)theta0=0*]]]) → tuple

yade._utils.**sumFacetNormalForces**(*(object)ids*[, *(int)axis=-1*]) → float

Sum force magnitudes on given bodies (must have *shape* of the *Facet* type), considering only part of forces perpendicular to each *facet's* face; if *axis* has positive value, then the specified axis (0=x, 1=y, 2=z) will be used instead of facet's normals.

yade._utils.**sumForces**(*(list)ids*, *(yade._minieigenHP.Vector3)direction*) → float

Return summary force on bodies with given *ids*, projected on the *direction* vector.

yade._utils.**sumTorques**(*(list)ids*, *(yade._minieigenHP.Vector3)axis*, *(yade._minieigenHP.Vector3)axisPt*) → float

Sum forces and torques on bodies given in *ids* with respect to axis specified by a point *axisPt* and its direction *axis*.

yade._utils.**totalForceInVolume**() → tuple

Return summed forces on all interactions and average isotropic stiffness, as tuple (Vector3,float)

yade._utils.**unbalancedForce**([*(bool)useMaxForce=False*]) → float

Compute the ratio of mean (or maximum, if *useMaxForce*) summary force on bodies and mean force magnitude on interactions. For perfectly static equilibrium, summary force on all bodies is zero (since forces from interactions cancel out and induce no acceleration of particles); this ratio will tend to zero as simulation stabilizes, though zero is never reached because of finite precision computation. Sufficiently small value can be e.g. 1e-2 or smaller, depending on how much equilibrium it should be.

yade._utils.**voidratio2D**([*(float)zlen=1*]) → float

Compute 2D packing void ratio $\frac{V-V_s}{V_s}$ where V is overall volume and V_s is volume of disks.

Parameters

zlen (*float*) – length in the third direction.

yade._utils.voxelPorosity([*(int)resolution=200*[, *(yade._minieigenHP.Vector3)start=Vector3(0, 0, 0)*[, *(yade._minieigenHP.Vector3)end=Vector3(0, 0, 0)*]]]) → float

Compute packing porosity $\frac{V-V_v}{V}$ where V is a specified volume (from start to end) and V_v is volume of voxels that fall inside any sphere. The calculation method is to divide whole volume into a dense grid of voxels (at given resolution), and count the voxels that fall inside any of the spheres. This method allows one to calculate porosity in any given sub-volume of a whole sample. It is properly excluding part of a sphere that does not fall inside a specified volume.

Parameters

- **resolution** (*int*) – voxel grid resolution, values bigger than resolution=1600 require a 64 bit operating system, because more than 4GB of RAM is used, a resolution=800 will use 500MB of RAM.
- **start** (Vector3) – start corner of the volume.
- **end** (Vector3) – end corner of the volume.

yade._utils.wireAll() → None

Set *Shape::wire* on all bodies to True, rendering them with wireframe only.

yade._utils.wireNoSpheres() → None

Set *Shape::wire* to True on non-spherical bodies (*Facets*, *Walls*).

yade._utils.wireNone() → None

Set *Shape::wire* on all bodies to False, rendering them as solids.

2.4.20 yade.ymport module

Import geometry from various formats ('import' is python keyword, hence the name 'ymport').

yade.ymport.blockMeshDict(*path*, *patchasWall=True*, *emptyasWall=True*, ***kw*)

Load openfoam's blockMeshDict file's "boundary" section as facets.

Parameters

- **path** (*str*) – file name. Typical value is: "system/blockMeshDict".
- **patchasWall** (*bool*) – load "patch"-es as walls.
- **emptyasWall** (*bool*) – load "empty"-es as walls.
- ****kw** – (unused keyword arguments) is passed to *utils.facet*

Returns

list of facets.

yade.ymport.ele(*nodeFileName*, *eleFileName*, *shift=(0, 0, 0)*, *scale=1.0*, ***kw*)

Import tetrahedral mesh from .ele file, return list of created tetrahedrons.

Parameters

- **nodeFileName** (*string*) – name of .node file
- **eleFileName** (*string*) – name of .ele file
- **shift** (*(float,float,float)*/Vector3) – (X,Y,Z) parameter moves the specimen.
- **scale** (*float*) – factor scales the given data.
- ****kw** – (unused keyword arguments) is passed to *utils.polyhedron*

yade.ymport.gengeo(*mntable*, *shift=Vector3(0, 0, 0)*, *scale=1.0*, ***kw*)

Imports geometry from LSMGenGeo library and creates spheres. Since 2012 the package is available in Debian/Ubuntu and known as python-demgengeo http://packages.qa.debian.org/p/python-demgengeo.html

Parameters

***mntable*: mntable**
object, which creates by LSMGenGeo library, see example

***shift*: [float,float,float]**
[X,Y,Z] parameter moves the specimen.

***scale*: float**
factor scales the given data.

****kw*: (unused keyword arguments)**
is passed to *utils.sphere*

LSMGenGeo library allows one to create pack of spheres with given [Rmin:Rmax] with null stress inside the specimen. Can be useful for Mining Rock simulation.

Example: examples/packs/packs.py, usage of LSMGenGeo library in examples/test/genCylLSM.py.

- https://answers.launchpad.net/esys-particle/+faq/877
- http://www.access.edu.au/lsmgengeo_python_doc/current/pythonapi/html/GenGeo-module.html
- https://svn.esscc.uq.edu.au/svn/esys3/lsm/contrib/LSMGenGeo/

`yade.ymport.gengeoFile`(*fileName='file.geo'*, *shift=Vector3(0, 0, 0)*, *scale=1.0*, *orientation=Quaternion((1, 0, 0), 0)*, ***kw*)

Imports geometry from LSMGenGeo .geo file and creates spheres. Since 2012 the package is available in Debian/Ubuntu and known as python-demgengeo http://packages.qa.debian.org/p/python-demgengeo.html

Parameters

***filename*: string**
file which has 4 colums [x, y, z, radius].

***shift*: Vector3**
Vector3(X,Y,Z) parameter moves the specimen.

***scale*: float**
factor scales the given data.

***orientation*: quaternion**
orientation of the imported geometry

****kw*: (unused keyword arguments)**
is passed to *utils.sphere*

Returns
list of spheres.

LSMGenGeo library allows one to create pack of spheres with given [Rmin:Rmax] with null stress inside the specimen. Can be useful for Mining Rock simulation.

Example: examples/packs/packs.py, usage of LSMGenGeo library in examples/test/genCylLSM.py.

- https://answers.launchpad.net/esys-particle/+faq/877
- http://www.access.edu.au/lsmgengeo_python_doc/current/pythonapi/html/GenGeo-module.html
- https://svn.esscc.uq.edu.au/svn/esys3/lsm/contrib/LSMGenGeo/

`yade.ymport.gmsh`(*meshfile='file.mesh'*, *shift=Vector3(0, 0, 0)*, *scale=1.0*, *orientation=Quaternion((1, 0, 0), 0)*, ***kw*)

Imports geometry from .mesh file and creates facets.

Parameters

***shift*: [float,float,float]**
[X,Y,Z] parameter moves the specimen.

***scale*: float**
factor scales the given data.

***orientation*: quaternion**
orientation of the imported mesh

****kw*: (unused keyword arguments)**
is passed to *utils.facet*

Returns
list of facets forming the specimen.

mesh files can easily be created with GMSH. Example added to examples/packs/packs.py

Additional examples of mesh-files can be downloaded from http://www-roc.inria.fr/gamma/download/download.php

`yade.ymport.gts`(*meshfile*, *shift=Vector3(0, 0, 0)*, *scale=1.0*, ***kw*)

Read given meshfile in gts format.

Parameters

***meshfile*: string**
name of the input file.

***shift*: [float,float,float]**
[X,Y,Z] parameter moves the specimen.

***scale*: float**
factor scales the given data.

****kw*: (unused keyword arguments)**
is passed to *utils.facet*

Returns
list of facets.

`yade.ymport.iges`(*fileName*, *shift=(0, 0, 0)*, *scale=1.0*, *returnConnectivityTable=False*, ***kw*)

Import triangular mesh from .igs file, return list of created facets.

Parameters

- **`fileName`** (*`string`*) – name of iges file
- **`shift`** (*`(float,float,float)`*/`Vector3`) – (X,Y,Z) parameter moves the specimen.
- **`scale`** (*`float`*) – factor scales the given data.
- **`**kw`** – (unused keyword arguments) is passed to *utils.facet*
- **`returnConnectivityTable`** (*`bool`*) – if True, apart from facets returns also nodes (list of (x,y,z) nodes coordinates) and elements (list of (id1,id2,id3) element nodes ids). If False (default), returns only facets

`yade.ymport.polyMesh`(*path*, *patchasWall=True*, *emptyasWall=True*, ***kw*)

Load openfoam's polyMesh directory as facets.

Parameters

- **`path`** (*`str`*) – directory path. Typical value is: "constant/polyMesh".
- **`patchAsWall`** (*`bool`*) – load "patch"-es as walls.
- **`emptyAsWall`** (*`bool`*) – load "empty"-es as walls.

- **kw – (unused keyword arguments) is passed to *utils.facet*

Returns
list of facets.

yade.ymport.stl(*file*, *dynamic=None*, *fixed=True*, *wire=True*, *color=None*, *highlight=False*, *noBound=False*, *material=-1*, *scale=1.0*, *shift=Vector3(0, 0, 0)*)

Import a .stl geometry in the form of a set of *Facet*-shaped bodies.

Parameters

- **file** (*string*) – the .stl file serving as geometry input
- **dynamic** (*bool*) – controls *Body.dynamic*
- **fixed** (*bool*) – controls *Body.dynamic* (with fixed = True imposing *Body.dynamic* = False) if *dynamic* attribute is not given
- **wire** (*bool*) – rendering option, passed to *Facet.wire*
- **color** – rendering option, passed to *Facet.color*
- **highlight** (*bool*) – rendering option, passed to *Facet.highlight*
- **noBound** (*bool*) – sets *Body.bounded* to False if True, preventing collision detection (and vice-versa)
- **material** – defines *material* properties, see *Defining materials* for usage
- **scale** (*float*) – scaling factor to e.g. dilate the geometry if > 1
- **shift** (Vector3) – for translating the geometry

Returns
a corresponding list of *Facet*-shaped bodies

yade.ymport.text(*fileName*, *shift=Vector3(0, 0, 0)*, *scale=1.0*, ***kw*)

Load sphere coordinates from file, returns a list of corresponding bodies; that may be inserted to the simulation with O.bodies.append().

Parameters

- **filename** (*string*) – file which has 4 colums [x, y, z, radius].
- **shift** (*[float,float,float]*) – [X,Y,Z] parameter moves the specimen.
- **scale** (*float*) – factor scales the given data.
- ****kw** – (unused keyword arguments) is passed to *utils.sphere*

Returns
list of spheres.

Lines starting with # are skipped

yade.ymport.textClumps(*fileName*, *shift=Vector3(0, 0, 0)*, *discretization=0*, *orientation=Quaternion((1, 0, 0), 0)*, *scale=1.0*, ***kw*)

Load clumps-members from file in a format selected by the `format` argument, insert them to the simulation.

Parameters

- **filename** (*str*) – file name
- **format** (*str*) – selected input format. Supported `'x_y_z_r'``(default), ``'x_y_z_r_clumpId'`
- **shift** (*[float,float,float]*) – [X,Y,Z] parameter moves the specimen.
- **scale** (*float*) – factor scales the given data.
- ****kw** – (unused keyword arguments) is passed to *utils.sphere*

Returns
list of spheres.

Lines starting with # are skipped

yade.ymport.textExt(*fileName*, *format='x_y_z_r'*, *shift=Vector3(0, 0, 0)*, *scale=1.0*, *attrs=[]*, ***kw*)

Load sphere coordinates from file in a format selected by the `format` argument, returns a list of corresponding bodies; that may be inserted to the simulation with `O.bodies.append()`.

Parameters

- **filename** (*str*) – file name
- **format** (*str*) – selected input format. Supported `'x_y_z_r'` (default), `'x_y_z_r_matId'`, `'id_x_y_z_r_matId'` (with `id` left unused), `'x_y_z_r_attrs'`; with whitespace delimiters in all cases
- **shift** (*[float,float,float]*) – [X,Y,Z] parameter moves the specimen.
- **scale** (*float*) – factor scales the given data.
- **attrs** (*list*) – attrs read from file if export.textExt(format='x_y_z_r_attrs') were used ('passed by reference' style)
- ****kw** – (unused keyword arguments) is passed to *utils.sphere*

Returns
list of spheres.

Lines starting with # are skipped

yade.ymport.textFacets(*fileName*, *format='x1_y1_z1_x2_y2_z2_x3_y3_z3'*, *shift=Vector3(0, 0, 0)*, *scale=1.0*, *attrs=[]*, ***kw*)

Load facet coordinates from file in a format selected by the `format` argument, returns a list of corresponding bodies; that may be inserted to the simulation with O.bodies.append().

Parameters

- **filename** (*str*) – file name
- **format** (*str*) – selected input format. Supported `'x1_y1_z1_x2_y2_z2_x3_y3_z3'``(default), ``'x1_y1_z1_x2_y2_z2_x3_y3_z3_matId'`, `'id_x1_y1_z1_x2_y2_z2_x3_y3_z3_matId'` or `'x1_y1_z1_x2_y2_z2_x3_y3_z3_attrs'`
- **shift** (*[float,float,float]*) – [X,Y,Z] parameter moves the specimen.
- **scale** (*float*) – factor scales the given data.
- **attrs** (*list*) – attrs read from file ('passed by reference' style)
- ****kw** – (unused keyword arguments) is passed to *utils.facet*

Returns
list of facets.

Lines starting with # are skipped

yade.ymport.textPolyhedra(*fileName*, *material*, *shift=Vector3(0, 0, 0)*, *scale=1.0*, *orientation=Quaternion((1, 0, 0), 0)*, ***kw*)

Load polyhedra from a text file.

Parameters

- **filename** (*str*) – file name. Expected file format is the one output by export.textPolyhedra.
- **shift** (*[float,float,float]*) – [X,Y,Z] parameter moves the specimen.

- **scale** (*float*) – factor scales the given data.
- **orientation** (*quaternion*) – orientation of the imported polyhedra
- ****kw** – (unused keyword arguments) is passed to *polyhedra_utils.polyhedra*

Returns
list of polyhedras.

Lines starting with # are skipped

yade.ymport.unv(*fileName*, *shift=(0, 0, 0)*, *scale=1.0*, *returnConnectivityTable=False*, ***kw*)

Import geometry from unv file, return list of created facets.

param string fileName
name of unv file

param (float,float,float)|Vector3 shift
(X,Y,Z) parameter moves the specimen.

param float scale
factor scales the given data.

param **kw
(unused keyword arguments) is passed to *utils.facet*

param bool returnConnectivityTable
if True, apart from facets returns also nodes (list of (x,y,z) nodes coordinates) and elements (list of (id1,id2,id3) element nodes ids). If False (default), returns only facets

unv files are mainly used for FEM analyses (are used by OOFEM and Abaqus), but triangular elements can be imported as facets. These files cen be created e.g. with open-source free software Salome.

Example: examples/test/unv-read/unvRead.py.

2.5 Installation

- Linux systems: Yade can be installed from *packages* (pre-compiled binaries) or *source code*. The choice depends on what you need: if you don't plan to modify Yade itself, package installation is easier. In the contrary case, you must download and install the source code.
- Other Operating Systems: Emulating Linux systems including Yade is proposed in this case, through *docker images* as well flash-drive or virtual machines images.
- 64 bit Operating Systems required; no support for 32 bit (i386).

2.5.1 Supported distributions and end-of-life policy

Yade follows the upstream support windows of Ubuntu and Debian: a release is dropped from our packages, Docker images and CI as soon as its *standard* (non-LTS-extension) support ends upstream.

- Ubuntu release cycle and support end dates: https://ubuntu.com/about/release-cycle
- Debian releases and their support status: https://www.debian.org/releases/

In practice this currently means Ubuntu releases older than 22.04 and Debian releases older than 13 (trixie) are no longer packaged, built as Docker images, or tested in CI.

2.5.2 Packages

Stable packages

Since 2011, all Ubuntu (starting from 11.10, Oneiric; and with the exception of Ubuntu 24.04 noble which requires using either daily packages or source code, see below) and Debian (starting from Wheezy)

versions have Yade in their main repositories. There are only stable releases in place. To install Yade, run the following:

```
sudo apt-get install yade
```

After that you can normally start Yade using the command `yade` or `yade-batch`.

This image shows versions and up to date status of Yade in some repositories.

Daily packages

Pre-built packages updated more frequently than the stable versions are provided for all currently supported Debian and Ubuntu versions.

These are "daily" versions of the packages which are being updated regularly and, hence, include all the newly added features.

For their installation, you need to add the yade-dem.org/packages repository to your /etc/apt/sources.list.

- Debian 13 **trixie** also with *high precision* `long double`, `float128` and `mpfr150` packages:

```
sudo bash -c 'echo "deb http://www.yade-dem.org/packages/ trixie main" >> /
↪etc/apt/sources.list.d/yadedaily.list'
```

- Debian 14 **forky** (current debian testing suite) also with *high precision* `long double`, `float128` and `mpfr150` packages:

```
sudo bash -c 'echo "deb http://www.yade-dem.org/packages/ forky main" >> /
↪etc/apt/sources.list.d/yadedaily.list'
```

- Ubuntu 22.04 **jammy** also with *high precision* `long double`, `float128` and `mpfr150` packages:

```
sudo bash -c 'echo "deb http://www.yade-dem.org/packages/ jammy main" >> /
↪etc/apt/sources.list.d/yadedaily.list'
```

- Ubuntu 24.04 **noble**:

```
sudo bash -c 'echo "deb http://www.yade-dem.org/packages/ noble main" >> /
↪etc/apt/sources.list.d/yadedaily.list'
```

- Ubuntu 26.04 **resolute**:

```
sudo bash -c 'echo "deb http://www.yade-dem.org/packages/ resolute main" >> /
↪etc/apt/sources.list.d/yadedaily.list'
```

Add the PGP-key AA915EEB as trusted and install `yadedaily`:

```
wget -O - http://www.yade-dem.org/packages/yadedev_pub.gpg | sudo tee /etc/apt/
↪trusted.gpg.d/yadedaily.asc
sudo apt-get update
sudo apt-get install yadedaily
```

After that you can normally start Yade using the command `yadedaily` or `yadedaily-batch`. `yadedaily` on older distributions can have some disabled features due to older library versions, shipped with particular distribution.

The Git-repository for packaging stuff is available on GitLab.

If you do not need `yadedaily`-package anymore, just remove the corresponding line in /etc/apt/sources.list and the package itself:

```
sudo apt-get remove yadedaily
```

To remove our key from keyring, execute the following command:

```
sudo apt-key remove AA915EEB
```

Daily and stable Yade versions can coexist without any conflicts, i.e., you can use `yade` and `yadedaily` at the same time.

2.5.3 Docker

Yade can be installed using docker images, which are daily built. Images contain both stable and daily versions of packages, except for Ubuntu 24.04 (see below). Docker images are based on supported distributions:

- Debian 13 **trixie**:

  ```
  docker run -it registry.gitlab.com/yade-dev/docker-prod:debian-trixie
  ```

- Ubuntu 22.04 **jammy**:

  ```
  docker run -it registry.gitlab.com/yade-dev/docker-prod:ubuntu22.04
  ```

- Ubuntu 24.04 **noble** (with only the *yadedaily* version):

  ```
  docker run -it registry.gitlab.com/yade-dev/docker-prod:ubuntu24.04
  ```

After the container is pulled and is running, Yade functionality can be checked:

```
yade --test
yade --check
yadedaily --test
yadedaily --check
```

2.5.4 Source code

Installation from source code is reasonable, when you want to add or modify constitutive laws, engines, functions etc., or use the recently added features, which are not yet available in packaged versions.

Doing so, we recommend to separate source-code-folder from a build-place-folder, where Yade will be configured and where the source code will be compiled. Here is an example for a folder structure:

```
myYade/                 ## base directory
        trunk/          ## folder for git-handled source code, see "Download" section␣
↪below
        build/          ## folder in which the sources will be compiled;␣
↪build-directory; use cmake here, see "Compilation.." sections below
        install/        ## install folder; contains the executables
```

Download

Installing from source, you can adopt either a release (numbered version, which is frozen) or the current development version (updated by the developers frequently). You should download the development version (called `trunk`) if you want to modify the source code, as you might encounter problems that will be fixed by the developers. Release versions will not be updated (except for updates due to critical and easy-to-fix bugs), but generally they are more stable than the trunk.

1. Releases can be downloaded from the download page, as compressed archive. Uncompressing the archive gives you a directory with the sources.
2. The development version (`trunk`) can be obtained from the code repository at GitLab.

We use GIT (the `git` command) for code management (install the `git` package on your system and create a GitLab account). From the top of of the above folder structure:

```
git clone git@gitlab.com:yade-dev/trunk.git
```

will download the whole code repository into `trunk` folder. Check out *Yade on GitLab* for more details on how to collaborate using `git`.

Alternatively, a read-only checkout is possible via https without a GitLab account (easier if you don't want to modify the trunk version):

```
git clone https://gitlab.com/yade-dev/trunk.git
```

For those behind a firewall, you can download the sources from our GitLab repository as compressed archive.

Release and trunk sources are compiled in exactly the same way.

Prerequisites

Yade compilation and execution rely on a number of mandatory and optional external softwares; they are checked before the compilation starts. Following dependencies are for instance mandatory:

- cmake build system
- gcc compiler (`g++`); other compilers will not work; you need g++>=4.2 for openMP support
- boost 1.47 or later
- Qt library
- freeglut3
- libQGLViewer
- python, numpy, ipython, sphinx
- matplotlib
- eigen algebra library (minimal required version 3.2.1)
- gdb debugger
- sqlite3 database engine

They can be installed from the command line of your Linux distribution, assuming you have root privileges.

For Ubuntu 22.04, 24.04, 26.04, **Debian 13** and their derivatives, just copy&paste to the terminal the following code block for installing all mandatory and optional dependencies (better avoid installing python packages with pip or conda):

```
sudo apt install cmake git freeglut3-dev libboost-all-dev fakeroot \
dpkg-dev build-essential g++ python3-dev python3-ipython python3-matplotlib \
libsqlite3-dev python3-numpy python3-tk gnuplot libgts-dev python3-pygraphviz \
libvtk9-dev libeigen3-dev python3-xlib python3-pyqt5 pyqt5-dev-tools python3-mpi4py \
python3-pyqt5.qtsvg libqglviewer-dev-qt5 \
python3-pil libjs-jquery python3-sphinx python3-git libxmu-dev libxi-dev libcgal-dev \
help2man libbz2-dev zlib1g-dev libopenblas-dev libsuitesparse-dev \
libmetis-dev python3-bibtexparser coinor-clp coinor-libclp-dev \
python3-mpmath libmpfr-dev libmpfrc++-dev libmpc-dev texlive-xetex python3-
↪pickleshare python3-ipython-genutils
```

Note

The GUI help/web view uses `QtWebKit` if available, falling back to `QtWebEngineWidgets` automatically otherwise. Neither is a hard dependency; install `python3-pyqt5.qtwebkit` (Debian 12/Ubuntu

24.04 and earlier) or `python3-pyqt5.qtwebengine` (Debian 13/14, Ubuntu 26.04, where `qtwebkit` binary packages were removed from the archive) if you want that feature.

If you are using other distributions than Debian or its derivatives you should install by yourself the software packages listed above. Their names in other distributions can differ from the names of the Debian-packages.

Some of the above packages are only required for some choice of Yade compilation options, for desired Yade features, in the subsequent `cmake` configuration of compilation. If a required package is eventually not installed the related features will be disabled automatically with a message appearing during `cmake` output (at the end, in particular). Generally speaking, it is advised to watch for notes and warnings/errors, which are shown by `cmake` in the following.

Compilation configuration

Then, inside the build-directory of the above folder structure, you should call `cmake` to configure the compilation process, passing a path to install folder (as an option) and the path to sources:

```
cmake -DCMAKE_INSTALL_PREFIX=../install ../trunk
```

In the above, note the `cmake -DOPTION1=VALUE1 -DOPTION2=VALUE2` syntax which is here applied to the lone `CMAKE_INSTALL_PREFIX` option, being part of a first group of `cmake` options that control the compilation process in itself or just slightly modify the behavior of the executable:

- CMAKE_INSTALL_PREFIX: path where Yade should be installed (/usr/local by default)
- CMAKE_VERBOSE_MAKEFILE: output additional information during compiling (OFF by default)
- CHOLMOD_GPU link Yade to custom SuiteSparse installation and activate GPU accelerated PFV, see *Accelerating Yade's FlowEngine with GPU* (OFF by default)
- DEBUG: compile in debug-mode, enabling a more convenient debugging or profiling by the user and leading to a much (1 or 2 orders of magnitude) slower executable (OFF by default)
- DISABLE_ALL: for switching off all available boolean options, before possibly enabling explicitely just some of them, e.g. `cmake -DDISABLE_ALL=ON -DENABLE_VTK=ON` (OFF by default)
- DISABLE_PKGS: comma-separated list of disabled packages i.e. names of source subdirectories under *pkg*, *preprocessing* or *postprocessing*, e.g. `cmake -DDISABLE_PKGS=fem,pfv,image`. If empty all packages will be built. The packages *common* and *dem* are required to run, but the project can be compiled without them. (EMPTY by default)
- ENABLE_ASAN: AddressSanitizer allows detection of memory errors, memory leaks, heap corruption errors and out-of-bounds accesses but it is slow (OFF by default)
- ENABLE_FAST_NATIVE: use maximum optimization compiler flags including `-Ofast` and `-mtune=native`. Note: `native` means that code will **only** run on the same processor type on which it was compiled. Observed speedup was 2% (below standard deviation measurement error) and above 5% if clang compiler was used. (OFF by default)
- ENABLE_OAR: generate a script for oar-based task scheduler, as discussed *here* (OFF by default)
- ENABLE_USEFUL_ERRORS: enable useful compiler errors which help a lot in error-free development (ON by default)
- LIBRARY_OUTPUT_PATH: path to install libraries (lib by default)
- MAX_LOG_LEVEL: *set maximum level* for LOG_* macros compiled with below `ENABLE_LOGGER`, (default is 5)
- NOSUFFIX: do not add a suffix after binary-name, see also `SUFFIX` option (OFF by default)
- PYTHON_VERSION: force Python version to the given one, e.g. `-DPYTHON_VERSION=3.5`. Set to -1 to automatically use the last version on the system (-1 by default)

- REAL_PRECISION_BITS, REAL_DECIMAL_PLACES: specify either of them to use a custom calculation precision of `Real` type. By default `double` (64 bits, 15 decimal places) precision is used as the `Real` type. See *high precision documentation* for additional details.
- runtimePREFIX: used for packaging, when install directory is not the same as runtime directory (/usr/local by default)
- SUFFIX: suffix, added after binary-names, see also `NOSUFFIX` option (version number by default)
- SUITESPARSEPATH: define this variable with the path to a custom suitesparse install
- USE_QT5: use Qt5 for GUI (ON by default)
- USE_QT6: use Qt6 for GUI. When enabled it overrides `USE_QT5` and requires Qt6, PyQt6 development tools, and a Qt6 build of libQGLViewer (OFF by default)
- VECTORIZE: enables vectorization and alignment in Eigen3 library, experimental (OFF by default)
- YADE_VERSION: explicitly set version number (is defined from git-directory by default)

As a more precise alternative to the above `DISABLE_*` options, other `cmake` options will select or unselect specific Yade classes for compilation, enabling or disabling additional Yade features while possibly requiring additional dependencies in form of external packages. They obey a `ENABLE_OPTION=ON` or `OFF` syntax as follows (see also the source code for a most up-to-date list):

- ENABLE_CGAL: enables a number of code sections using the CGAL library, requires `libcgal-dev` package (ON by default)
- ENABLE_COMPLEX_MP: use boost multiprecision complex and mpc (as an extension to MPFR, see `ENABLE_MPFR`) for `ComplexHP<N>`, otherwise use `std::complex<RealHP<N>>`. See *high precision documentation* for additional details. Requires `libmpc-dev` (ON by default if possible: requires boost >= 1.71)
- ENABLE_DEFORM: enable the constant volume deformation approach for bodies [Haustein2017] (OFF by default)
- ENABLE_FEMLIKE: enable FEM-like meshed solids (ON by default)
- ENABLE_GL2PS: enable GL2PS-option (ON by default)
- ENABLE_GTS: enable GTS-option (ON by default)
- ENABLE_GUI: enable a Qt GUI. Defaults to Qt5 and can be switched to Qt6 with `USE_QT6` (ON by default)
- ENABLE_LBMFLOW: enable LBM computations, e.g. the use of *HydrodynamicsLawLBM* (ON by default)
- ENABLE_LS_DEM: enable a *LevelSet* shape description (ON by default)
- ENABLE_LINSOLV: enable the use of optimized algebra libraries SuiteSparse (sparse algebra, requires eigen>=3.1), OpenBLAS (optimized and parallelized alternative to the standard blas+lapack) and Metis (matrix preconditioning) for the optional fluid coupling *FlowEngine*, see `ENABLE_PFVFLOW` below. Requires `libopenblas-dev libsuitesparse-dev libmetis-dev` packages (ON by default)
- ENABLE_LIQMIGRATION: enable LIQMIGRATION-option, see [Mani2013] for details (OFF by default)
- ENABLE_LOGGER: provides *logging* possibilities for each class thanks to boost::log library. See also `MAX_LOG_LEVEL` in the above (ON by default)
- ENABLE_MASK_ARBITRARY: enable arbitrary precision of bitmask variables (only `Body::groupMask` yet implemented) (experimental). If ON, use -DMASK_ARBITRARY_SIZE=int to set number of used bits (256 by default) (OFF by default)

- ENABLE_MPFR: use mpfr in `C++` and mpmath in `python` for higher precision `Real` or for CGAL exact predicates, see *high precision documentation* for more details. Requires `python3-mpmath libmpfr-dev libmpfrc++-dev` packages (OFF by default)
- ENABLE_MPI: enable MPI environment and communication thanks to OpenMPI and python3-mpi4py (see also there), for parallel distributed computing (distributed memory) and Yade-OpenFOAM coupling. Requires `python3-mpi4py` (ON by default)
- ENABLE_MULTI_REAL_HP: allow using twice, quadruple or higher precisions of `Real` as `RealHP<2>`, `RealHP<4>` or `RealHP<N>` in computationally demanding sections of `C++` code. See *high precision documentation* for additional details (ON by default).
- ENABLE_OPENMP: enable OpenMP-parallelizing of Yade execution (ON by default)
- ENABLE_PARTIALSAT : enable the partially saturated clay engine *PartialSatClayEngine*, under construction (ON by default)
- ENABLE_PFVFLOW: enable PFV *FlowEngine*, with package requirements as per ENABLE_LINSOLV (ON by default)
- ENABLE_POTENTIAL_BLOCKS: enable *PotentialBlock* shape description thanks for instance to the COIN-OR Linear Programming Solver, requires `coinor-clp coinor-libclp-dev libopenblas-dev` (ON by default)
- ENABLE_POTENTIAL_PARTICLES: enable *PotentialParticle* shape description, requires `libopenblas-dev` (ON by default)
- ENABLE_PROFILING: enable profiling, e.g., shows some more metrics, which can define bottlenecks of the code (OFF by default)
- ENABLE_SPH: enable Smoothed Particle Hydrodynamics (OFF by default)
- ENABLE_THERMAL : enable *ThermalEngine* (ON by default, experimental)
- ENABLE_TWOPHASEFLOW: enable *TwoPhaseFlowEngine* (ON by default)
- ENABLE_VTK: enable exports of data using the VTK library, e.g. *VTKRecorder* engine, requires a `libvtk*-dev` (e.g., `libvtk9-dev` on Ubuntu 22.04) package (ON by default)

Maintaining a consistent choice for options values, in addition to using the same version of source code, is often necessary for successfully reloading previous Yade saves, see *O.load*.

For using more extended parameters of cmake, please follow the corresponding documentation on https://cmake.org/documentation.

Compilation and usage

If cmake finishes without errors, you will see all enabled and disabled options at the end. Then start the actual compilation process with:

```
make
```

The compilation process can take a considerable amount of time, be patient. If you are using a multi-core system you can use the parameter `-j` to speed-up the compilation and split the compilation onto many cores. For example, on 4-core machines it would be reasonable to set the parameter `-j4`. Note, Yade requires approximately 3GB RAM per core for compilation, otherwise the swap-file will be used and compilation time dramatically increases.

The installation is performed with the following command:

```
make install
```

The `install` command will in fact also recompile if source files have been modified. Hence there is no absolute need to type the two commands separately. You may receive make errors if you don't have permission to write into the target folder. These errors are not critical but without writing permissions Yade won't be installed in /usr/local/bin/.

After the compilation finished successfully, the new built can be started by navigating to /path/to/installfolder/bin and calling yade via (based on version yade-2014-02-20.git-a7048f4):

```
cd /path/to/installfolder/bin
./yade-2014-02-20.git-a7048f4
```

For building the documentation you should at first execute the command `make install` and then `make doc` to build it, provided that package `texlive-xetex` is present. On some multi-language systems an error `Building format(s) --all. This may take some time... fmtutil failed.` may occur, in that case a package `locales-all` is required.

The generated files will be stored in your current install directory /path/to/installfolder/share/doc/yade-your-version. Once again writing permissions are necessary for installing into /usr/local/share/doc/. To open your local documentation go into the folder html and open the file index.html with a browser.

`make manpage` command generates and moves manpages in a standard place. `make check` command executes standard test to check the functionality of the compiled program.

Yade can be compiled not only by GCC-compiler, but also by CLANG front-end for the LLVM compiler. For that you set the environment variables CC and CXX upon detecting the C and C++ compiler to use:

```
export CC=/usr/bin/clang
export CXX=/usr/bin/clang++
cmake -DOPTION1=VALUE1 -DOPTION2=VALUE2
```

Clang does not support OpenMP-parallelizing for the moment, that is why the feature will be disabled.

Supported linux releases

Currently supported[1] linux releases for source code installation and their respective docker files are:

- Ubuntu 22.04
- Ubuntu 24.04
- Ubuntu 26.04
- Debian 13 trixie
- Debian 14 forky (current testing)
- openSUSE 15
- Arch Linux

Ubuntu and Debian releases older than the ones listed above are dropped once their upstream standard support ends; see *Supported distributions and end-of-life policy* for the policy and links to the upstream release schedules.

The above links include the bash commands used to prepare the linux distribution and environment for installing and testing Yade. During Yade development workflow, these instructions are automatically performed using the gitlab continuous integration service after each merge to the master branch, which makes sure that Yade always works correctly on these linux distributions. In fact the above installation procedure closely corresponds to following step by step these instructions in following order:

1. Bash commands in the respective Dockerfile to install necessary packages,
2. do `git clone https://gitlab.com/yade-dev/trunk.git`, just like in the previous *Download* paragraph
3. then the `cmake_*` commands in the .gitlab-ci.yml file for respective distribution, corresponding to previous *Compilation configuration* paragraph
4. then the `make_*` commands to compile yade, as per *Compilation and usage*

[1] To see details of the latest build log click on the *master* branch.

5. and finally the `--check` and `--test` commands.
6. Optionally documentation can be built with `make doc` command, as also explained in *Compilation and usage*. However currently it is not guaranteed to work on all linux distributions due to frequent interface changes in sphinx.

These instructions use `ccache` and `ld.gold` to *speed-up compilation* as described below.

Python 2 backward compatibility

Following the end of Python 2 support (beginning of 2020), Yade compilation on a Python 2 ecosystem is no longer garanteed since the 6e097e95 trunk version. Python 2-compilation of the latter is still possible using the above `PYTHON_VERSION` cmake option, requiring Python 2 version of prerequisites packages whose list can be found in the corresponding paragraph (Python 2 backward compatibility) of the historical doc.

Ongoing development of Yade now assumes a Python 3 environment, and you may refer to some notes about *converting Python 2 scripts into Python 3* if needed.

2.5.5 Speed-up compilation

Compile with ccache

Caching previous compilations with ccache can significantly speed up re-compilation:

```
cmake -DCMAKE_CXX_COMPILER_LAUNCHER=ccache [options as usual]
```

Additionally one can check current ccache status with command `ccache --show-stats` (`ccache -s` for short) or change the default cache size stored in file `~/.ccache/ccache.conf`.

Compile with distcc

When spliting the compilation on many cores (`make -jN`), `N` is limited by the available cores and memory. It is possible to use more cores if remote computers are available, distributing the compilation with distcc (see distcc documentation for configuring slaves and master):

```
export CC="distcc gcc"
export CXX="distcc g++"
cmake [options as usual]
make -jN
```

The two tools can be combined, adding to the above exports:

```
export CCACHE_PREFIX="distcc"
```

Compile with cmake UNITY_BUILD

This option concatenates source files in batches containing several `*.cpp` each, in order to share the overhead of include directives (since most source files include the same boost headers, typically). It accelerates full compilation from scratch (quite significantly). It is activated by adding the following to cmake command, `CMAKE_UNITY_BUILD_BATCH_SIZE` defines the maximum number of files to be concatenated together (the higher the better, main limitation might be available RAM):

```
-DCMAKE_UNITY_BUILD=ON -DCMAKE_UNITY_BUILD_BATCH_SIZE=18
```

This method is helpless for incremental re-compilation and might even be detrimental since a full batch has to be recompiled each time a single file is modified. If it is anticipated that specific files will need incremental compilation they can be excluded from the unity build by assigning their full path to cmake flag `NO_UNITY` (a single file or a comma-separated list):

```
-DCMAKE_UNITY_BUILD=ON -DCMAKE_UNITY_BUILD_BATCH_SIZE=18 -DNO_UNITY=../trunk/pkg/dem/
↪CohesiveFrictionalContactLaw.cpp
```

Alternatively inside a C++ file place the following comment, to mark it as skipped in the unity build:

```
// SKIP_UNITY_BUILD
```

Link time

The link time can be reduced by changing the default linker from `ld` to `ld.gold`. They are both in the same package `binutils` (on opensuse15 it is package `binutils-gold`). To perform the switch execute these commands as root:

```
ld --version
update-alternatives --install "/usr/bin/ld" "ld" "/usr/bin/ld.gold" 20
update-alternatives --install "/usr/bin/ld" "ld" "/usr/bin/ld.bfd" 10
ld --version
```

To switch back run the commands above with reversed priorities `10  20`. Alternatively a manual selection can be performed by command: `update-alternatives --config ld`.

Note: `ld.gold` is incompatible with the compiler wrapper `mpicxx` in some distributions, which is manifested as an error in the `cmake` stage. We do not use `mpicxx` for our gitlab builds currently. If you want to use it then disable `ld.gold`. Cmake `MPI`-related failures have also been reported without the `mpicxx` compiler, if it happens then the only solution is to disable either `ld.gold` or the `MPI` feature.

2.5.6 Cloud Computing

It is possible to exploit cloud computing services to run Yade. The combo Yade/Amazon Web Service has been found to work well, namely. Detailed instructions for migrating to amazon can be found in the section *Using YADE with cloud computing on Amazon EC2*.

2.5.7 GPU Acceleration

The FlowEngine can be accelerated with CHOLMOD's GPU accelerated solver. The specific hardware and software requirements are outlined in the section *Accelerating Yade's FlowEngine with GPU*.

2.5.8 Special builds

The software can be compiled by a special way to find some specific bugs and problems in it: memory corruptions, data races, undefined behaviour etc.

The listed sanitizers are runtime-detectors. They can only find the problems in the code, if the particular part of the code is executed. If you have written a new C++ class (constitutive law, engine etc.) try to run your Python script with the sanitized software to check, whether the problem in your code exist.

AddressSanitizer

AddressSanitizer is a memory error detector, which helps to find heap corruptions, out-of-bounds errors and many other memory errors, leading to crashes and even wrong results.

To compile Yade with this type of sanitizer, use ENABLE_ASAN option:

```
cmake -DENABLE_ASAN=1
```

The compilation time, memory consumption during build and the size of build-files are much higher than during the normall build. Monitor RAM and disk usage during compilation to prevent out-of-RAM problems.

To find the proper libasan library in your particular distribution, use `locate` or `find /usr -iname "libasan*so"` command. Then, launch your yade executable in connection with that libasan library, e.g.:

```
LD_PRELOAD=/some/path/to/libasan.so yade
```

By default the leak detector is enabled in the asan build. Yade is producing a lot of leak warnings at the moment. To mute those warnings and concentrate on other memory errors, one can use detect_leaks=0 option. Accounting for the latter, the full command to run tests with the AddressSanitized-Yade on Debian 10 Buster is:

```
ASAN_OPTIONS=detect_leaks=0:verify_asan_link_order=false yade --test
```

If you add a new check script, it is being run automatically through the AddressSanitizer in the CI-pipeline.

2.5.9 Yubuntu

If you are not running a Linux system there is a way to create an Ubuntu live-usb on any usb mass-storage device (minimum size 10GB). It is a way to boot the computer on a linux system with Yadedaily pre-installed without affecting the original system. More informations about this alternative are available here (see the README file first). Note that the images there date back from 2018 and use ubuntu16.04, for newer versions of yade see below.

Alternatively, images of a linux virtual machine can be downloaded here (ubuntu20.04) , or for older (ubuntu16.04) versions here. They should run on any system with a virtualization software (tested with VirtualBox and VMWare).

2.6 Acknowledging Yade

We kindly ask YADE users to cite the documentation in scientific publications as a way to assess YADE's contribution to their field.

For the YADE project in general, please reference the documentation as a whole:

- V. Šmilauer et al. (2021), *Yade Documentation* 3rd ed. The Yade Project. DOI:10.5281/zenodo.5705394

```
@article{yade2021,
  title        = {Yade Documentation (3rd edition)},
  author       = {Smilauer, Vaclav and Angelidakis, Vasileios and Catalano,␣
↪Emanuele and Caulk, Robert and Chareyre, Bruno and Ch{\ifmmode\grave{e}\else\`
↪{e}\fi}vremont, William and Dorofeenko, Sergei and Duriez, J{\ifmmode\acute{e}\
↪else\'{e}\fi}r{\ifmmode\hat{o}\else\^{o}\fi}me and Dyck, Nolan and Elias, Jan␣
↪and Er, Burak and Eulitz, Alexander and Gladky, Anton and Guo, Ning and Jakob,␣
↪Christian and Kneib, Francois and Kozicki, Janek and Marzougui, Donia and␣
↪Maurin, Raphael and Modenese, Chiara and Pekmezi, Gerald and Scholt{\
↪ifmmode\grave{e}\else\`{e}\fi}s, Luc and Sibille, Luc and Stransky, Jan and␣
↪Sweijen, Thomas and Thoeni, Klaus and Yuan, Chao},
  year         = {2021},
  journal      = {Zenodo},
  doi          = {10.5281/zenodo.5705394}
}
```

For newer features of YADE, please reference the review article:

- V. Angelidakis et al. "YADE - An extensible framework for the interactive simulation of multi-scale, multiphase, and multiphysics particulate systems", *Computer Physics Communications* 304 (2024): 109293. DOI:10.1016/j.cpc.2024.109293

```
@article{yade2024,
  author  = {Angelidakis, Vasileios and Boschi, Katia and Brzezi{\ifmmode\acute
↪{n}\else\'{n}\fi}ski, Karol and Caulk, Robert A. and Chareyre, Bruno and del␣
↪Valle, Carlos Andr{\ifmmode\acute{e}\else\'{e}\fi}s and Duriez, J{\
↪ifmmode\acute{e}\else\'{e}\fi}r{\ifmmode\hat{o}\else\^{o}\fi}me and Gladky,␣
↪Anton and van der Haven, Dingeman L. H. and Kozicki, Janek and Pekmezi, Gerald␣
↪and Scholt{\ifmmode\grave{e}\else\`{e}\fi}s, Luc and Thoeni, Klaus},
  title   = {{YADE - An extensible framework for the interactive simulation of␣
↪multiscale, multiphase, and multiphysics particulate systems}},
  journal = {Computer Physics Communications},
  volume  = {304},
  pages   = {109293},
  year    = {2024},
  month   = nov,
  issn    = {0010-4655},
  publisher = {North-Holland},
  doi     = {10.1016/j.cpc.2024.109293},
  url     = {https://doi.org/10.1016/j.cpc.2024.109293}
}
```

Ideally, users cite both documents to acknowledge the work of both earlier and later developers of YADE. Beyond acknowledgment, proper referencing helps find new use cases and new users by tracking the citations on Yade's Scholar profile.

Chapter 3

Yade for programmers

3.1 Programmer's manual

3.1.1 Build system

Yade uses cmake the cross-platform, open-source build system for managing the build process. It takes care of configuration, compilation and installation. CMake is used to control the software compilation process using simple platform and compiler independent configuration files. CMake generates native makefiles and workspaces that can be used in the compiler environment of your choice.

Building

The structure of Yade source tree is presented below. We shall call each top-level component *module* (excluding, `doc`, `examples` and `scripts` which don't participate in the build process). Some subdirectories of *modules* are skipped for brevity, see `README.rst` files therein for more information:

```
cMake/            ## cmake files used to detect compilation requirements
core/             ## core simulation building blocks
data/             ## data files used by yade, packaged separately
doc/              ## this documentation
examples/         ## examples directory
gui/              ## user interfaces
   qt5/              ## same, but for qt5
lib/              ## support libraries, not specific to simulations
preprocessing/    ## files associated with creation or generation of the simulation
   dem/              ## creating a DEM simulation
   potential/        ## creating a PotentialBlocks or PotentialParticles simulation
   README.rst        ## more information about this directory
pkg/              ## simulation-specific files
   common/           ## generally useful classes
   dem/              ## classes for Discrete Element Method
   README.rst        ## more information about this directory
postprocessing/   ## files associated with extracting results for postprocessing
   dem/              ## general data extraction from DEM, no particular data target
   image/            ## creating images from simulation
   vtk/              ## extracting data for VTK
   README.rst        ## more information about this directory
py/               ## python modules
scripts/          ## helper scripts including packaging and checks-and-tests
```

Header installation

CMAKE uses the original source layout and it is advised to use `#include <module/Class.hpp>` style of inclusion rather than `#include "Class.hpp"` even if you are in the same directory. The following table gives a few examples:

Original header location	Included as
`core/Scene.hpp`	`#include <core/Scene.hpp>`
`lib/base/Logging.hpp`	`#include <lib/base/Logging.hpp>`
`lib/serialization/Serializable.hpp`	`#include <lib/serialization/Serializable.hpp>`
`pkg/dem/SpherePack.hpp`	`#include <pkg/dem/SpherePack.hpp>`

Automatic compilation

In the `pkg/` directory, situation is different. In order to maximally ease addition of modules to yade, all `*.cpp` files are *automatically scanned recursively* by CMAKE and considered for compilation.

To enable/disable some component use the cmake flags `ENABLE_FEATURE`, which are listed in:

1. *compilation instructions*.
2. CMakeLists.txt.

When some component is enabled an extra `#define` flag `YADE_FEATURE` is passed from cmake to the compiler. Then inside the code both the `.cpp` and `.hpp` files which contain the `FEATURE` feature should have an `#ifdef YADE_FEATURE` guard at the beginning.

Linking

The order in which modules might depend on each other is given as follows:

module	resulting shared library	dependencies
lib	`libyade-support.so`	can depend on external libraries, may **not** depend on any other part of Yade.
core	`libcore.so`	`yade-support`; *may* depend on external libraries.
pkg	`libplugins.so`	`core`, `yade-support`
gui	`libQtGUI.so`, `libPythonUI.so`	`lib`, `core`, `pkg`
py	(many files)	`lib`, `core`, `pkg`, external

3.1.2 Development tools

Integrated Development Environment and other tools

A frequently used IDE is Kdevelop. We recommend using this software for navigating in the sources, compiling and debugging. Other useful tools for debugging and profiling are Valgrind and KCachegrind. A series of wiki pages is dedicated to these tools in the development section of the wiki.

Yade is agnostic to the IDE used; it can be compiled and run directly from the command line. You can modify the source code using any text editor, such as vim https://www.vim.org/, emacs https://www.gnu.org/software/emacs/, vscode https://code.visualstudio.com/, or any other editor of your choice.

Hosting and versioning

The Yade project is kindly hosted at GitLab:

- source code on gitlab
- issue and bug tracking on gitlab
- release downloads on GitLab
- questions and answers on GitLab

The versioning software used is GIT, for which a short tutorial can be found in *Yade on GitLab*. GIT is a distributed revision control system. It is available packaged for all major linux distributions.

The repository can be http-browsed.

Development process

Git is used for version control. The main development branch is called `master` and is hosted at GitLab. For the development process, the following steps are recommended:

1. Clone the repository to your local machine: `git clone https://gitlab.com/yade-dev/trunk.git`
2. Create a new branch for your work: `git checkout -b my-new-feature`
3. Make your changes and commit them: `git commit -am 'Add some feature'`
4. Push to the branch: `git push origin my-new-feature`
5. Submit a merge request on GitLab: Merge Request

The merge request will be reviewed by the developers and, if accepted, merged into the main branch. Yade has a wide range of pipelines that are automatically triggered by GitLab when a new commit is pushed to the repository. These pipelines include building the software, running tests, and generating the documentation. The results of these pipelines can be viewed on the GitLab CI/CD page or by clicking on the green checkmark next to a commit in the GitLab interface. If some tests fail, the developers will be notified and the merge request will not be accepted until the issues are resolved.

It is required to add at least one line into the ChangeLog file in the root directory of the repository for each merge request. This file is used to generate the release notes for each new version of Yade.

How to make a release

The release process is automated using GitLab CI/CD pipelines. The release process is triggered by creating a new tag in the repository. The tag should be named according to the semver convention, (e.g., `2025.2.0`), whre the first number is the year, the second number is the month, and the third number is the patch version. The release process will build the software, run tests, and generate the documentation.

1. Create RELEASE file in the root folder with the version number in it.
2. Update Changelog file, put the proper date and version number in the top of the file.
3. Create branch using the following command and format:

```
git checkout -b YYYY.M.0
```

4. Tag release “git tag -as YYYY.M.0 -m”YYYY.M.0”
5. Return to master branch and remove RELEASE file
6. Push master, new branch and tags to gitlab
7. Download tar.gz
8. Create asc-file (signature): `gpg --armor --sign --detach-sig tarball.tar.gz`

RELEASE file should contain the version number in the following format:

```
YYYY.MM.0
```

where `YYYY` is the year and `MM` is the month of the release. For example, the release file for the February 2025 release should contain the following text:

```
2025.2.0
```

Build robot

A build robot hosted at UMS Gricad is tracking source code changes via gitlab pipeline mechanism. Each time a change in the source code is committed to the main development branch via GIT, or a Merge Request (MR) is submitted the "buildbot" downloads and compiles the new version, and then starts a series of tests.

If a compilation error has been introduced, it will be notified to the yade-dev mailing list and to the committer, thus helping to fix problems quickly. If the compilation is successful, the buildbot starts unit regression tests and "check tests" (see below) and report the results. If all tests are passed, a new version of the documentation is generated and uploaded to the website in html and pdf formats. As a consequence, those two links always point to the documentation (the one you are reading now) of the last successful build, and the delay between commits and documentation updates are very short (minutes). The buildbot activity and logs can be browsed online.

The output of each particular build is directly accessible by clicking the green "Passed" button, and then clicking "Browse" in the "Job Artifacts" on the right.

3.1.3 Debugging

For yade debugging two tools are available:

1. Use the debug build so that the stack trace provides complete information about potential crash. This can be achieved in two ways:
   a) Compiling yade with cmake option `-DDEBUG=ON`,
   b) Installing `yade-dbgsym` debian/ubuntu package (this option will be available after this task is completed).
2. Use *Logging* framework described below.

These tools can be used in conjunction with other software. A detailed discussion of these is on yade wiki. These tools include: kdevelop, valgrind, alleyoop, kcachegrind, ddd, gdb, kompare, kdiff3, meld.

Note

On some linux systems stack trace will not be shown and a message `ptrace: Operation not permitted` will appear instead. To enable stack trace issue command: `sudo echo 0 > /proc/sys/kernel/yama/ptrace_scope`. To disable stack trace issue command `sudo echo 1 > /proc/sys/kernel/yama/ptrace_scope`.

Hint

When debugging make sure there is enough free space in /tmp.

Logging

Yade uses boost::log library for flexible logging levels and per-class debugging. See also description of *log module*. A cmake compilation option `-DENABLE_LOGGER=ON` must be supplied during compilation[1].

[1] Without `-DENABLE_LOGGER=ON` cmake option the debug macros in /lib/base/Logging.hpp use regular `std::cerr` for output, per-class logging and log levels do not work.

```
In [1]: import log

In [2]: yade.log.setLevel("_log.cpp",5)
<INFO> _log.cpp:101 void setLevel(std::__cxx11::string, int): filter log level for _log.cpp has been set to 5

In [3]: log.setLevel("NewtonIntegrator",4)
<INFO> _log.cpp:101 void setLevel(std::__cxx11::string, int): filter log level for NewtonIntegrator has been set to 4

In [4]: log.getUsedLevels()
Out[4]: {'Default': 3, 'NewtonIntegrator': 4, '_log.cpp': 5}

In [5]: yade.log.testAllLevels()
<NOFILTER> :54 void testAllLevels(): Test log level: LOG_0_NOFILTER, test int: 0
<FATAL ERROR> _log.cpp:55 void testAllLevels(): Test log level: LOG_1_FATAL, test int: 1
<ERROR> _log.cpp:56 void testAllLevels(): Test log level: LOG_2_ERROR, test int: 2
<WARNING> _log.cpp:57 void testAllLevels(): Test log level: LOG_3_WARN, test int: 3
<INFO> _log.cpp:58 void testAllLevels(): Test log level: LOG_4_INFO, test int: 4
<DEBUG> _log.cpp:59 void testAllLevels(): Test log level: LOG_5_DEBUG, test int: 5
<NOFILTER> :62 void testAllLevels(): Below 6 variables are printed at filter level TRACE, then macro TRACE; is used

In [6]:
```

Figure *imgLogging* shows example use of logging framework. Usually a `ClassName` appears in place of `_log.cpp` shown on the screenshot. It is there because the `yade.log` module uses `CREATE_CPP_LOCAL_LOGGER` macro instead of the regular `DECLARE_LOGGER` and `CREATE_LOGGER`, which are *discussed below*.

Note

Default format of log message is:

```
<severity level> ClassName:LineNumber FunctionName: Log Message
```

special macro `LOG_NOFILTER` is printed without `ClassName` because it lacks one.

Config files can be saved and loaded via *readConfigFile* and *saveConfigFile*. The *defaultConfigFileName* is read upon startup if it exists. The filter level setting `-f` supplied from command line will override the setting in config file.

Log levels

Following debug levels are supported:

Table 1: Yade logging verbosity levels.

macro name	filter name	option	explanation
`LOG_NOFILTER`	`log.NOFILTER`	`-f0`	Will print only the unfiltered messages. The LOG_-NOFILTER macro is for developer use only, so basically `-f0` means that nothing will be printed. This log level is not useful unless a very silent mode is necessary.
`LOG_FATAL`	`log.FATAL`	`-f1`	Will print only critical errors. Even a throw to yade python interface will not recover from this situation. This is usually followed by yade exiting to shell.
`LOG_ERROR`	`log.ERROR`	`-f2`	Will also print errors which do not require to throw to yade python interface. Calculations will continue, but very likely the results will be all wrong.
`LOG_WARN`	`log.WARN`	`-f3`	Will also print warnings about recoverable problems that you should be notified about (e.g., invalid value in a configuration file, so yade fell back to the default value).
`LOG_INFO`	`log.INFO`	`-f4`	Will also print all informational messages (e.g. something was loaded, something was called, etc.).
`LOG_DEBUG`	`log.DEBUG`	`-f5`	Will also print debug messages. A yade developer puts them everywhere, and yade user enables them on *per-class basis* to provide some extra debug info.
`LOG_TRACE`	`log.TRACE`	`-f6`	Trace messages, they capture every possible detail about yade behavior.

Yade default log level is `yade.log.WARN` which is the same as invoking `yade -f3`.

Setting a filter level

Warning

The messages (such as `a << b << " message."`) given as arguments to `LOG_*` macros are used only if the message passes the filter level. **Do not use such messages to perform mission critical calculations**.

There are two settings for the filter level, the `Default` level used when no `ClassName` (or `"filename.cpp"`) specific filter is set and a filter level set for specific `ClassName` (or `"filename.cpp"`). They can be set with following means:

1. When starting yade with `yade -fN` command, where `N` sets the `Default` filter level. The default value is `yade.log.WARN` (3).
2. To change `Default` filter level during runtime invoke command `log.setLevel("Default",value)` or `log.setDefaultLogLevel(value)`:

```
Yade [1]: import log

Yade [2]: log.setLevel("Default",log.WARN)

Yade [3]: log.setLevel("Default",3)

Yade [4]: log.setDefaultLogLevel(log.WARN)

Yade [5]: log.setDefaultLogLevel(3)
```

3. To change filter level for `SomeClass` invoke command:

```
Yade [6]: import log

Yade [7]: log.setLevel("NewtonIntegrator",log.TRACE)

Yade [8]: log.setLevel("NewtonIntegrator",6)
```

4. To change the filter level for `"filename.cpp"` use the name specified when creating it. For example manipulating filter log level of `"_log.cpp"` might look like following:

```
Yade [9]: import log

Yade [10]: log.getUsedLevels()
Out[10]: {'Default': 3, 'NewtonIntegrator': 6}

Yade [11]: log.setLevel("_log.cpp",log.WARN)

Yade [12]: log.getUsedLevels()
Out[12]: {'Default': 3, 'NewtonIntegrator': 6, '_log.cpp': 3}

Yade [13]: log.getAllLevels()["_log.cpp"]
Out[13]: 3
```

Debug macros

To enable debugging for particular class the `DECLARE_LOGGER;` macro should be put in class definition inside header to create a separate named logger for that class. Then the `CREATE_LOGGER(ClassName);` macro must be used in the class implementation `.cpp` file to create the static variable. Sometimes a logger is necessary outside the class, such named logger can be created inside a `.cpp` file and by convention its name should correspond to the name of the file, use the macro `CREATE_CPP_LOCAL_LOGGER("filename.cpp");` for this. On rare occasions logging is necessary inside `.hpp` file outside of a class (where the local class named logger is unavailable), then the solution is to use `LOG_NOFILTER(…)` macro, because it is the only one that can work without a named logger. If the need arises this solution can be improved, see Logging.cpp for details.

All debug macros (`LOG_TRACE`, `LOG_DEBUG`, `LOG_INFO`, `LOG_WARN`, `LOG_ERROR`, `LOG_FATAL`, `LOG_NOFILTER`) listed in section above accept the `std::ostream` syntax inside the brackets, such as `LOG_TRACE( a << b << " text" )`. The `LOG_NOFILTER` is special because it is always printed regardless of debug level, hence it should be used only in development branches.

Additionally seven macros for printing variables at `LOG_TRACE` level are available: `TRVAR1`, `TRVAR2`, `TRVAR3`, `TRVAR4`, `TRVAR5`, `TRVAR6` and `TRVARn`. They print the variables, e.g.: `TRVAR3(testInt,testStr,testReal);` or `TRVARn((testInt)(testStr)(testReal))`. See function testAllLevels for example use.

The macro `TRACE;` prints a `"Been here"` message at `TRACE` log filter level, and can be used for quick debugging.

Utility debug macros

The `LOG_TIMED_*` family of macros:

In some situations it is useful to debug variables inside a **very fast**, or maybe a **multithreaded**, loop. In such situations it would be useful to:

1. Avoid spamming console with very fast printed messages and add some print timeout to them, preferably specified with units of seconds or milliseconds.
2. Make sure that each separate thread has opportunity to print message, without interleaving such messages with other threads.

To use above functionality one must `#include <lib/base/LoggingUtils.hpp>` in the `.cpp` file which provides the `LOG_TIMED_*` and `TIMED_TRVAR*` macro family. Example usage can be found in function

testTimedLevels.

To satisfy the first requirement all `LOG_TIMED_*` macros accept **two arguments**, where the first argument is the wait timeout, using standard C++14 / C++20 time units, example use is `LOG_TIMED_INFO( 2s , "test int: " << testInt++);` to print every 2 seconds. But only seconds and milliseconds are accepted (this can be changed if necessary).

To satisfy the second requirement a thread_local static Timer variable is used. This way each thread in a parallel loop can print a message every `500ms` or `10s` e.g. in this parallel loop. The time of last print to console is stored independently for each thread and an extra code block which checks time is added. It means that a bit more checks are done than typical `LOG_*` which only perform an integer comparison to check filter level. Therefore suggested use is only during heavy debugging. When debugging is finished then better to remove them.

Note

The `*_TRACE` family of macros are removed by compiler during the release builds, because the default `-DMAX_LOG_LEVEL` is 5. So those are very safe to use, but to have them working locally make sure to compile yade with `cmake -DMAX_LOG_LEVEL=6` option.

The `LOG_ONCE_*` family of macros:

In a similar manner a `LOG_ONCE_*` and `ONCE_TRVAR*` family of macros is provided inside file LoggingUtils.hpp. Then the message is printed only once.

All debug macros are summarized in the table below:

Table 2: Yade debug macros.

macro name	explanation
`DECLARE_LOGGER;`	Declares logger variable inside class definition in `.hpp` file.
`CREATE_LOGGER(ClassName);`	Creates logger static variable (with name `"ClassName"`) inside class implementation in `.cpp` file.
`TEMPLATE_CREATE_LOGGER(ClassName<OtherCla`	Creates logger static variable (with name `"ClassName<OtherClass>"`) inside class implementation in a `.cpp` file. Use this for templated classes.
`CREATE_CPP_LOCAL_LOGGER("filename.cpp");`	Creates logger static variable outside of any class (with name `"filename.cpp"`) inside the `filename.cpp` file.
`LOG_TRACE`, `LOG_TIMED_TRACE`, `LOG_ONCE_TRACE`, `LOG_DEBUG`, `LOG_TIMED_DEBUG`, `LOG_ONCE_DEBUG`, `LOG_INFO`, `LOG_TIMED_INFO`, `LOG_ONCE_INFO`, `LOG_WARN`, `LOG_TIMED_WARN`, `LOG_ONCE_WARN`, `LOG_ERROR`, `LOG_TIMED_ERROR`, `LOG_ONCE_ERROR`, `LOG_FATAL`, `LOG_TIMED_FATAL`, `LOG_ONCE_FATAL`, `LOG_NOFILTER`, `LOG_TIMED_NOFILTER`, `LOG_ONCE_NOFILTER`	Prints message using `std::ostream` syntax, like: `LOG_TRACE( a << b << " text" )` `LOG_TIMED_TRACE( 5s , a << b << " text" );` , prints every 5 seconds `LOG_TIMED_DEBUG( 500ms , a );`, prints every 500 milliseconds `LOG_ONCE_TRACE( a << b << " text" );` , prints just once `LOG_ONCE_DEBUG( a );`, prints only once

`TRVAR1`, `TIMED_TRVAR1`, `ONCE_TRVAR1`, `TRVAR2`, `TIMED_TRVAR2`, `ONCE_TRVAR2`, `TRVAR3`, `TIMED_TRVAR3`, `ONCE_TRVAR3`, `TRVAR4`, `TIMED_TRVAR4`, `ONCE_TRVAR4`, `TRVAR5`, `TIMED_TRVAR5`, `ONCE_TRVAR5`, `TRVAR6`, `TIMED_TRVAR6`, `ONCE_TRVAR6`, `TRVARn`, `TIMED_TRVARn`, `ONCE_TRVARn`	Prints provided variables like:
	`TRVAR3(testInt,testStr,testReal);`
	`TRVARn((testInt)(testStr)(testReal));`
	`TIMED_TRVAR3( 10s , testInt , testStr , testReal);`
	`ONCE_TRVARn( (testInt)(testStr)(testReal));`
	See file py/_log.cpp for example use.

`TRACE;`	Prints a `"Been here"` message at `TRACE` log filter level.
`LOG_TIMED_6`, `LOG_6_TRACE`, `LOG_ONCE_6`, `LOG_TIMED_5`, `LOG_5_DEBUG`, `LOG_ONCE_5`, `LOG_TIMED_4`, `LOG_4_INFO`, `LOG_ONCE_4`, `LOG_TIMED_3`, `LOG_3_WARN`, `LOG_ONCE_3`, `LOG_TIMED_2`, `LOG_2_ERROR`, `LOG_ONCE_2`, `LOG_TIMED_1`, `LOG_1_FATAL`, `LOG_ONCE_1`, `LOG_TIMED_0`, `LOG_0_NOFILTER`,	Additional macro aliases for easier use in editors with tab completion. They have have a filter level number in their name.

Maximum log level

Using boost::log for log filtering means that each call to `LOG_*` macro must perform a single integer comparison to determine if the message passes current filter level. For production use calculations should be as fast as possible and this filtering is not optimal, because the macros are *not optimized out*, as they can be re-enabled with a simple call to `log.setLevel("Default",log.TRACE)` or `log.setLevel("Default",6)`. The remedy is to use the cmake compilation option `MAX_LOG_LEVEL=4` (or 3) which will remove macros higher than the specified level during compilation. The code will run slightly faster and the command `log.setLevel("Default",6)` will only print a warning that such high log level (which can be checked with *log.getMaxLevel()* call) is impossible to obtain with current build.

Note

At the time when logging was introduced into yade the speed-up gain was so small, that it turned out to be impossible to measure with `yade -f0 --stdperformance` command. Hence this option `MAX_LOG_LEVEL` was introduced only on principle.

The upside of this approach is that yade can be compiled in a non-debug build, and the log filtering framework can be still used.

3.1.4 Regression tests

Yade contains two types of regression tests, some are unit tests while others are testing more complex simulations. Although both types can be considered regression tests, the usage is that we name the first simply "regression tests", while the latest are called "check tests". Both series of tests can be ran at yade startup by passing the options "test" or "checkall"

```
yade --test
yade --checkall
```

The `yade --checkall` is a complete check. To skip checks lasting more than 30 seconds one can use this command

```
yade --check
```

Unit regression tests

Unit regression tests are testing the output of individual functors and engines in well defined conditions. They are defined in the folder py/tests/. The purpose of unit testing is to make sure that the behaviour of the most important classes remains correct during code development. Since they test classes one by one, unit tests can't detect problems coming from the interaction between different engines in a typical simulation. That is why check tests have been introduced.

To add a new test, the following steps must be performed:

1. Place a new file such as py/tests/dummyTest.py.
2. Add the file name such as `dummyTest` to the py/tests/___init___.py file.
3. If necessary modify the `import` and `allModules` lines in py/tests/___init___.py.
4. According to instructions in python unittest documentation use commands such as `self.assertTrue(…)`, `self.assertFalse(…)` or `self.assertRaises(…,…)` to report possible errors.

Note

It is important that all variables used in the test are stored inside the class (using the `self.` accessor), and that all preparations are done inside the function `setUp()`.

Check tests

Check tests (also see README) perform comparisons of simulation results between different versions of yade, as discussed here. They differ with regression tests in the sense that they simulate more complex situations and combinations of different engines, and usually don't have a mathematical proof (though there is no restriction on the latest). They compare the values obtained in version N with values obtained in a previous version or any other "expected" results. The reference values must be hardcoded in the script itself or in data files provided with the script. Check tests are based on regular yade scripts, so that users can easily commit their own scripts to trunk in order to get some automatized testing after commits from other developers.

When check fails the script should return an error message via python command `raise YadeCheckError(messageString)` telling what went wrong. If the script itself fails for some reason and can't generate an output, the log will contain only "scriptName failure". If the script defines differences on obtained and awaited data, it should print some useful information about the problem. After this occurs, the automatic test will stop the execution with error message.

An example dummy check test scripts/checks-and-tests/checks/checkTestDummy.py demonstrates a minimal empty test. A little more functional example check test can be found in scripts/checks-and-tests/checks/checkTestTriax.py. It shows results comparison, output, and how to define the path to data files using `checksPath`. Users are encouraged to add their own scripts into the scripts/checks-and-tests/checks/ folder. Discussion of some specific checktests design in questions and answers is welcome. Note that *re-compiling* is required before the newly added scripts can be launched by `yade --check` (or direct changes have to be performed in "lib" subfolders). A check test should never need more than a few seconds to run. If your typical script needs more, try to reduce the number of elements or the number of steps.

To add a new check, the following steps must be performed:

1. Place a new file such as scripts/checks-and-tests/checks/checkTestDummy.py,
2. Inside the new script use `checksPath` when it is necessary to load some data file, like scripts/checks-and-tests/checks/data/100spheres
3. When error occurs raise exception with command `raise YadeCheckError(messageString)`

It is recommended to run simulation for certain number of steps before checking the condition (`O.run(Nsteps,wait = True)`) rather than pausing simulation with `O.pause()`. The latter may cause segmentation fault during checks (related to execfile limitation described in the warning below).

Warning

Due to the limitation of execfile the local variables created in one check script are passed down to the check scripts executed after it. Hence creating a local variable in one script called e.g. `Body` will break the scripts executed after it, when they will try to create a new `Body()`. The workaround is to use unique non-trivial variable names in the check scripts.

GUI Tests

In order to add a new GUI test one needs to add a file to scripts/checks-and-tests/gui directory. File must be named according to the following convention: `testGuiName.py` with an appropriate test `Name` in place (the `testGui.sh` script is searching for files matching this pattern). The scripts/checks-and-tests/gui/testGuiBilliard.py may serve as a boilerplate example. The important "extra" parts of the code (taken from e.g. example directory) are:

1. `from testGuiHelper import TestGUIHelper`
2. `scr = TestGUIHelper("Billiard")`, make sure to put the chosen test `Name` in place of `Billiard`.
3. Establish a reasonable value of `guiIterPeriod` which makes the test finish in less than 30 seconds.
4. Inside `O.engines` there has to be a call at the end of the loop to `PyRunner(iterPeriod=guiIterPeriod, command='scr.screenshotEngine()')`.

5. The last command in the script should be `O.run(guiIterPeriod * scr.getTestNum() + 1)` to start the test process.
6. Make sure to push to yade-data repository the reference screenshots (for dealing with `./data` dir see *Yade on GitLab*). These screenshots can be also obtained from artifacts by clicking "Download" button in the gitlab pipeline, next to the "Browse" button in the right pane.

These tests can be run locally, after adjusting the paths at the start of testGui.sh script. Two modes of operation are possible:

1. Launch on the local desktop via command: `scripts/checks-and-tests/gui/testGui.sh`, in this case the screenshots will be different from those used during the test.
2. Or launch inside a virtual xserver via command: `xvfb-run -a -s "-screen 0 1600x1200x24" scripts/checks-and-tests/gui/testGui.sh`, then the screenshots will be similar to those used in the test, but still there may be some differences in the font size. In such case it is recommended to use the reference screenshots downloaded from the artifacts in the gitlab pipeline (see point 6. above).

Care should be taken to not use random colors of bodies used in the test. Also no windows such as 3d View or Inspector view should be opened in the script `testGuiName.py`, because they are opened during the test by the `TestGUIHelper` class.

Note

It is not possible to call GUI tests from a call such as `yade --test` because of the necessity to launch YADE inside a virtual xserver.

3.1.5 Conventions

The following coding rules should be respected; documentation is treated separately.

- general
  - C++ source files have `.hpp` and `.cpp` extensions (for headers and implementation, respectively). In rare cases `.ipp` is used for pure template code.
  - All header files should have the `#pragma once` multiple-inclusion guard.
  - Do not type `using namespace …` in header files, this can lead to obscure bugs due to namespace pollution.
  - Avoid `using std::something` in `.hpp` files. Feel free to use them as much as you like inside `.cpp` files. But remember that the usual problems with this practice still apply: wrong type or function might be used instead of the one that you would expect. But since it's limited to a single `.cpp` file, it will be easier to debug and the convenience might outweight the associated dangers.
  - Use tabs for indentation. While this is merely visual in `C++`, it has semantic meaning in python; inadvertently mixing tabs and spaces can result in syntax errors.
- capitalization style
  - Types should be always capitalized. Use CamelCase for composed class and typenames (`GlobalEngine`). Underscores should be used only in special cases, such as functor names.
  - Class data members and methods must not be capitalized, composed names should use lowercase camelCase (`glutSlices`). The same applies for functions in python modules.
  - Preprocessor macros are uppercase, separated by underscores; those that are used outside the core take (with exceptions) the form `YADE_*`, such as *YADE_CLASS_BASE_DOC_* macro family*.
- programming style

- Be defensive, if it has no significant performance impact. Use assertions abundantly: they don't affect performance (in the optimized build) and make spotting error conditions much easier.
- Use `YADE_CAST` and `YADE_PTR_CAST` where you want type-check during debug builds, but fast casting in optimized build.
- Initialize all class variables in the default constructor. This avoids bugs that may manifest randomly and are difficult to fix. Initializing with NaN's will help you find otherwise unitialized variable. (This is taken care of by *YADE_CLASS_BASE_DOC_* macro family* macros for user classes)

Using clang-format

The file .clang-format contains the config which should produce always the same results. It works with `clang-format --version` >= 10. The aim is to eliminate commits that change formatting. The script scripts/clang-formatter.sh can be invoked on either file or a directory and will do the reformatting. Usually this can be integrated with the editor, see clang-format documentation (except that for vim `py3f` command has to be used), and in kdevelop it is added as a custom formatter.

The script scripts/python-formatter.sh applies our coding conventions to formatting of python scripts. It should be used before committing changes to python scripts.

For more help see:

1. clang-format documentation
2. yapf3 documentation

Sometimes it is useful to disable formatting in a small section of the file. In order to do so, put the guards around this section:

1. In `C++` use:

```
// clang-format off
.....
// clang-format on
```

2. In `Python` use:

```
# yapf: disable
....
# yapf: enable
```

Class naming

Although for historical reasons the naming scheme is not completely consistent, these rules should be obeyed especially when adding a new class.

GlobalEngines* and *PartialEngines
: GlobalEngines should be named in a way suggesting that it is a performer of certain action (like *ForceResetter*, *InsertionSortCollider*, *Recorder*); if this is not appropriate, append the `Engine` to the characteristics name (e.g. *GravityEngine*). *PartialEngines* have no special naming convention different from *GlobalEngines*.

Dispatchers
: Names of all dispatchers end in `Dispatcher`. The name is composed of type it creates or, in case it doesn't create any objects, its main characteristics. Currently, the following dispatchers[2] are defined:

[2] Not considering OpenGL dispatchers, which might be replaced by regular virtual functions in the future.

dispatcher	arity	dispatch types	created type	functor type	functor prefix
BoundDispatcher	1	*Shape*	*Bound*	*BoundFunctor*	`Bo1`
IGeomDispatcher	2 (symetric)	2 × *Shape*	*IGeom*	*IGeomFunctor*	`Ig2`
IPhysDispatcher	2 (symetric)	2 × *Material*	*IPhys*	*IPhysFunctor*	`Ip2`
LawDispatcher	2 (asymetric)	*IGeom* *IPhys*	*(none)*	*LawFunctor*	`Law2`

Respective abstract functors for each dispatchers are *BoundFunctor*, *IGeomFunctor*, *IPhysFunctor* and *LawFunctor*.

Functors

Functor name is composed of 3 parts, separated by underscore.

1. prefix, composed of abbreviated functor type and arity (see table above)
2. Types entering the dispatcher logic (1 for unary and 2 for binary functors)
3. Return type for functors that create instances, simple characteristics for functors that don't create instances.

To give a few examples:

- *Bo1_Sphere_Aabb* is a *BoundFunctor* which is called for *Sphere*, creating an instance of *Aabb*.
- *Ig2_Facet_Sphere_ScGeom* is binary functor called for *Facet* and *Sphere*, creating and instace of *ScGeom*.
- *Law2_ScGeom_CpmPhys_Cpm* is binary functor (*LawFunctor*) called for types *ScGeom (Geom)* and *CpmPhys*.

Documentation

Documenting code properly is one of the most important aspects of sustained development.

Read it again.

Most code in research software like Yade is not only used, but also read, by developers or even by regular users. Therefore, when adding new class, always mention the following in the documentation:

- purpose
- details of the functionality, unless obvious (algorithms, internal logic)
- limitations (by design, by implementation), bugs
- bibliographical reference, if using non-trivial published algorithms (see below)
- references to other related classes
- hyperlinks to bugs, blueprints, wiki or mailing list about this particular feature.

As much as it is meaningful, you should also

- update any other documentation affected
- provide a simple python script demonstrating the new functionality in `scripts/test`.

Sphinx documentation

Most c++ classes are wrapped in Python, which provides good introspection and interactive documentation (try writing `Material?` in the ipython prompt; or `help(CpmState)`).

Syntax of documentation is ReST (reStructuredText, see reStructuredText Primer). It is the same for c++ and python code.

- Documentation of c++ classes exposed to python is given as 3rd argument to *YADE_CLASS_BASE_DOC_* macro family* introduced below.
- Python classes/functions are documented using regular python docstrings. Besides explaining functionality, meaning and types of all arguments should also be documented. Short pieces of code might be very helpful. See the *utils* module for an example.

Note

Use C++ string literal when writing docstrings in C++. By convention the `R"""(raw text)"""` is used. For example see *here* and here.

Note

Remember that inside C++ docstrings it is possible to invoke python commands which are executed by yade when documentation is being compiled. For example compare this source docstring with the *final effect*.

In addition to standard ReST syntax, yade provides several shorthand macros:

`:yref:`
creates hyperlink to referenced term, for instance:

```
:yref:`CpmMat`
```

becomes *CpmMat*; link name and target can be different:

```
:yref:`Material used in the CPM model<CpmMat>`
```

yielding *Material used in the CPM model*.

`:ysrc:`
creates hyperlink to file within the source tree (to its latest version in the repository), for instance core/Cell.hpp. Just like with `:yref:`, alternate text can be used with

```
:ysrc:`Link text<target/file>`
```

like this. This cannot be used to link to a specified line number, since changing the file will cause the line numbers to become outdated. To link to a line number use `:ysrccommit:` described below.

`:ysrccommit:`
creates hyperlink to file within the source tree at the specified commit hash. This allows to link to the line numbers using for example `#L121` at the end of the link. Use it just like the `:ysrc:` except that commit hash must be provided at the beginning:

```
:ysrccommit:`Link text<commithash/target/file#Lnumber>`

:ysrccommit:`default engines<775ae7436/py/__init__.py.in#L112>`
```

becomes default engines.

Linking to `inheritanceGraph*`
: To link to an inheritance graph of some base class a *global anchor* is created with name `inheritanceGraph*` added in front of the class name, for example `:ref:`Shape<inheritanceGraphShape>`` yields link to *inheritance graph of Shape*.

`|ycomp|`
: is used in attribute description for those that should not be provided by the user, but are auto-computed instead; `|ycomp|` expands to *(auto-computed)*.

`|yupdate|`
: marks attributes that are periodically updated, being subset of the previous. `|yupdate|` expands to *(auto-updated)*.

`$...$`
: delimits inline math expressions; they will be replaced by:

```
:math:`...`
```

and rendered via LaTeX. To write a single dollar sign, escape it with backslash `\$`.

Displayed mathematics (standalone equations) can be inserted as explained in Math support for HTML outputs in Sphinx.

As a reminder in the standard ReST syntax the references are:

`:ref:`
: is the the standard restructured text reference to an anchor placed elsewere in the text. For instance an anchor `.. _NumericalDamping:` is placed in formulation.rst then it is linked to with `:ref:`NumericalDamping`` inside the source code.

`.. _anchor-name:`
: is used to place anchors in the text, to be referenced from elsewhere in the text. Symbol `_` is forbidden in the anchor name, because it has a special meaning: `_anchor` specifies anchor, while `anchor_` links to it, see below.

`anchor-name_`
: is used to place a link to anchor within the same file. It is a shorter form compared to the one which works between different files: `:ref:`. For example usage on anchor `imgQtGui` see here and here.

Note

The command `:scale: NN %` (with percent) does not work well with `.html` + `.pdf` output, better to specify `:width: NN cm`. Then it is the same size in `.html` and `.pdf`.. For example see here which becomes *this picture*. But bear in mind that maximum picture width in `.pdf` is `16.2 cm`.

Bibliographical references

As in any scientific documentation, references to publications are very important. To cite an article, first add it in BibTeX format to files doc/references.bib or doc/yade-*.bib depending whether that reference used Yade (the latter cases) or not (the former). Please adhere to the following conventions:

1. Keep entries in the form `Author2008` (`Author` is the first author), `Author2008b` etc if multiple articles from one author;
2. Try to fill mandatory fields for given type of citation;
3. Do not use `\'{i}` funny escapes for accents, since they will not work with the HTML output; put everything in straight utf-8.

In your docstring, the `Author2008` article can be then cited by `[Author2008]_`; for example:

```
According to [Allen1989]_, the integration scheme …
```

will be rendered as

> According to [Allen1989], the integration scheme …

Separate class/function documentation

Some c++ might have long or content-rich documentation, which is rather inconvenient to type in the c++ source itself as string literals. Yade provides a way to write documentation separately in py/_extraDocs.py file: it is executed after loading c++ plugins and can set `__doc__` attribute of any object directly, overwriting docstring from c++. In such (exceptional) cases:

1. Provide at least a brief description of the class in the c++ code nevertheless, for people only reading the code.
2. Add notice saying "This class is documented in detail in the py/_extraDocs.py file".
3. Add documentation to py/_extraDocs.py in this way:

```
module.YourClass.__doc__='''
        This is the docstring for YourClass.

        Class, methods and functions can be documented this way.

        .. note:: It can use any syntax features you like.

'''
```

Note

Boost::python embeds function signatures in the docstring (before the one provided by the user). Therefore, before creating separate documentation of your function, have a look at its `__doc__`-attribute and copy the first line (and the blank line afterwards) in the separate docstring. The first line is then used to create the function signature (arguments and return value).

Internal c++ documentation

doxygen was used for automatic generation of c++ code. Since user-visible classes are defined with sphinx now, it is not meaningful to use doxygen to generate overall documentation. However, take care to document well internal parts of code using regular comments, including public and private data members.

3.1.6 Support framework

Besides the framework provided by the c++ standard library (including STL), boost and other dependencies, Yade provides its own specific services.

Pointers

Shared pointers

Yade makes extensive use of shared pointers `shared_ptr`.[3] Although it probably has some performance impacts, it greatly simplifies memory management, ownership management of c++ objects in python and so forth. To obtain raw pointer from a `shared_ptr`, use its `get()` method; raw pointers should be used in case the object will be used only for short time (during a function call, for instance) and not stored anywhere.

[3] Either `boost::shared_ptr` or `tr1::shared_ptr` is used, but it is always imported with the `using` statement so that unqualified `shared_ptr` can be used.

Python defines thin wrappers for most c++ Yade classes (for all those registered with *YADE_CLASS_BASE_DOC_* macro family* and several others), which can be constructed from `shared_ptr`; in this way, Python reference counting blends with the `shared_ptr` reference counting model, preventing crashes due to python objects pointing to c++ objects that were destructed in the meantime.

Typecasting

Frequently, pointers have to be typecast; there is choice between static and dynamic casting.

- `dynamic_cast` (`dynamic_pointer_cast` for a `shared_ptr`) assures cast admissibility by checking runtime type of its argument and returns NULL if the cast is invalid; such check obviously costs time. Invalid cast is easily caught by checking whether the pointer is NULL or not; even if such check (e.g. `assert`) is absent, dereferencing NULL pointer is easily spotted from the stacktrace (debugger output) after crash. Moreover, `shared_ptr` checks that the pointer is non-NULL before dereferencing in debug build and aborts with "Assertion 'px!=0' failed." if the check fails.
- `static_cast` is fast but potentially dangerous (`static_pointer_cast` for `shared_ptr`). Static cast will return non-NULL pointer even if types don't allow the cast (such as casting from `State*` to `Material*`); the consequence of such cast is interpreting garbage data as instance of the class cast to, leading very likely to invalid memory access (segmentation fault, "crash" for short).

To have both speed and safety, Yade provides 2 macros:

`YADE_CAST`
: expands to `static_cast` in optimized builds and to `dynamic_cast` in debug builds.

`YADE_PTR_CAST`
: expands to `static_pointer_cast` in optimized builds and to `dynamic_pointer_cast` in debug builds.

Basic numerics

The floating point type to use in Yade is `Real`, which is by default typedef for `double` (64 bits, 15 decimal places).[4]

Yade uses the Eigen library for computations. It provides classes for 2d and 3d vectors, quaternions and 3x3 matrices templated by number type; their specialization for the `Real` type are typedef'ed with the "r" suffix, and occasionally useful integer types with the "i" suffix:

- `Vector2r`, `Vector2i`
- `Vector3r`, `Vector3i`
- `Quaternionr`
- `Matrix3r`

Yade additionally defines a class named *Se3r*, which contains spatial position (`Vector3r Se3r::position`) and orientation (`Quaternionr Se3r::orientation`), since they are frequently used one with another, and it is convenient to pass them as single parameter to functions.

Eigen provides full rich linear algebra functionality. Some code further uses the [cgal] library for computational geometry.

In Python, basic numeric types are wrapped and imported from the `yade.minieigenHP` module; the types drop the `r` type qualifier at the end, the syntax is otherwise similar. `Se3r` is not wrapped at all, only converted automatically, rarely as it is needed, from/to a `(Vector3,Quaternion)` tuple/list. See *high precision section* for more details.

```
# cross product
Yade [14]: Vector3(1,2,3).cross(Vector3(0,0,1))
Out[14]: Vector3(2,-1,0)
```

(continues on next page)

[4] See *high precision documentation* for additional details.

(continued from previous page)

```
# construct quaternion from axis and angle
Yade [15]: Quaternion(Vector3(0,0,1),pi/2)
Out[15]: Quaternion((0,0,1),1.570796326794896558)
```

Note

Quaternions are internally stored as 4 numbers. Their usual human-readable representation is, however, (normalized) axis and angle of rotation around that axis, and it is also how they are input/output in Python. Raw internal values can be accessed using the `[0]` … `[3]` element access (or `.W()`, `.X()`, `.Y()` and `.Z()` methods), in both c++ and Python.

Run-time type identification (RTTI)

Since serialization and dispatchers need extended type and inheritance information, which is not sufficiently provided by standard RTTI. Each yade class is therefore derived from `Factorable` and it must use macro to override its virtual functions providing this extended RTTI:

`YADE_CLASS_BASE_DOC(Foo,Bar Baz,"Docstring")` creates the following virtual methods (mediated via the `REGISTER_CLASS_AND_BASE` macro, which is not user-visible and should not be used directly):

- `std::string getClassName()` returning class name (`Foo`) as string. (There is the `typeid(instanceOrType).name()` standard c++ construct, but the name returned is compiler-dependent.)
- `unsigned getBaseClassNumber()` returning number of base classes (in this case, 2).
- `std::string getBaseClassName(unsigned i=0)` returning name of i-th base class (here, `Bar` for i=0 and `Baz` for i=1).

Warning

RTTI relies on virtual functions; in order for virtual functions to work, at least one virtual method must be present in the implementation (`.cpp`) file. Otherwise, virtual method table (vtable) will not be generated for this class by the compiler, preventing virtual methods from functioning properly.

Some RTTI information can be accessed from python:

```
Yade [16]: yade.system.childClasses('Shape')
Out[16]:
{'Box',
 'ChainedCylinder',
 'Clump',
 'Cylinder',
 'DeformableCohesiveElement',
 'DeformableElement',
 'Facet',
 'FluidDomainBbox',
 'GridConnection',
 'GridNode',
 'LevelSet',
 'Lin4NodeTetra',
 'Lin4NodeTetra_Lin4NodeTetra_InteractionElement',
 'Node',
 'PFacet',
 'Polyhedra',
 'PotentialBlock',
```

(continues on next page)

(continued from previous page)

```
 'PotentialParticle',
 'Sphere',
 'Subdomain',
 'Tetra',
 'Wall'}

Yade [17]: Sphere().__class__.__name__              ## getClassName()
Out[17]: 'Sphere'
```

Serialization

Serialization serves to save simulation to file and restore it later. This process has several necessary conditions:

- classes know which attributes (data members) they have and what are their names (as strings);
- creating class instances based solely on its name;
- knowing what classes are defined inside a particular shared library (plugin).

This functionality is provided by 3 macros and 4 optional methods; details are provided below.

`Serializable::preLoad`, `Serializable::preSave`, `Serializable::postLoad`, `Serializable::postSave`
: Prepare attributes before serialization (saving) or deserialization (loading) or process them after serialization or deserialization.

  See *Attribute registration*.

`YADE_CLASS_BASE_DOC_*`
: Inside the class declaration (i.e. in the `.hpp` file within the `class Foo { /* … */};` block). See *Attribute registration*.

  Enumerate class attributes that should be saved and loaded; associate each attribute with its literal name, which can be used to retrieve it. See *YADE_CLASS_BASE_DOC_* macro family*.

  Additionally documents the class in python, adds methods for attribute access from python, and documents each attribute.

`REGISTER_SERIALIZABLE`
: In header file, but *after* the class declaration block. See *Class factory*.

  Associate literal name of the class with functions that will create its new instance (`ClassFactory`).

  Must be declared inside `namespace yade`.

`YADE_PLUGIN`
: In the implementation `.cpp` file. See *Plugin registration*.

  Declare what classes are declared inside a particular plugin at time the plugin is being loaded (yade startup).

  Must be declared inside `namespace yade`.

Attribute registration

All (serializable) types in Yade are one of the following:

- Type deriving from *Serializable*, which provide information on how to serialize themselves via overriding the `Serializable::registerAttributes` method; it declares data members that should be serialzed along with their literal names, by which they are identified. This method then invokes `registerAttributes` of its base class (until `Serializable` itself is reached); in this way, derived classes properly serialize data of their base classes.

This funcionality is hidden behind the macro *YADE_CLASS_BASE_DOC_* macro family* used in class declaration body (header file), which takes base class and list of attributes:

```
YADE_CLASS_BASE_DOC_ATTRS(ThisClass,BaseClass,"class documentation",
↪((type1,attribute1,initValue1,,"Documentation for attribute 1
↪"))((type2,attribute2,initValue2,,"Documentation for attribute 2")));
```

Note that attributes are encoded in double parentheses, not separated by commas. Empty attribute list can be given simply by `YADE_CLASS_BASE_DOC_ATTRS(ThisClass,BaseClass,"documentation",)` (the last comma is mandatory), or by omiting `ATTRS` from macro name and last parameter altogether.

- Fundamental type: strings, various number types, booleans, `Vector3r` and others. Their "handlers" (serializers and deserializers) are defined in `lib/serialization`.
- Standard container of any serializable objects.
- Shared pointer to serializable object.

Yade uses the excellent boost::serialization library internally for serialization of data.

Note

`YADE_CLASS_BASE_DOC_ATTRS` also generates code for attribute access from python; this will be *discussed later*. Since this macro serves both purposes, the consequence is that attributes that are serialized can always be accessed from python.

Yade also provides callback for before/after (de) serialization, virtual functions *Serializable::preProcessAttributes* and *Serializable::postProcessAttributes*, which receive one `bool deserializing` argument (`true` when deserializing, `false` when serializing). Their default implementation in *Serializable* doesn't do anything, but their typical use is:

- converting some non-serializable internal data structure of the class (such as multi-dimensional array, hash table, array of pointers) into a serializable one (pre-processing) and fill this non-serializable structure back after deserialization (post-processing); for instance, InteractionContainer uses these hooks to ask its concrete implementation to store its contents to a unified storage (`vector<shared_ptr<Interaction> >`) before serialization and to restore from it after deserialization.
- precomputing non-serialized attributes from the serialized values; e.g. *Facet* computes its (local) edge normals and edge lengths from vertices' coordinates.

Class factory

Each serializable class must use `REGISTER_SERIALIZABLE`, which defines function to create that class by `ClassFactory`. `ClassFactory` is able to instantiate a class given its name (as string), which is necessary for deserialization.

Although mostly used internally by the serialization framework, programmer can ask for a class instantiation using `shared_ptr<Factorable> f=ClassFactory::instance().createShared("ClassName");`, casting the returned `shared_ptr<Factorable>` to desired type afterwards. *Serializable* itself derives from `Factorable`, i.e. all serializable types are also factorable.

Note

Both macros `REGISTER_SERIALIZABLE` and `YADE_PLUGIN` have to be declared inside yade namespace.

Plugin registration

Yade loads dynamic libraries containing all its functionality at startup. ClassFactory must be taught about classes each particular file provides. `YADE_PLUGIN` serves this purpose and, contrary to *YADE_CLASS_BASE_DOC_* macro family*, must be placed in the implementation (.cpp) file, inside yade namespace. It simply enumerates classes that are provided by this file:

```
YADE_PLUGIN((ClassFoo)(ClassBar));
```

Note

You must use parentheses around the class name even if there is only one class (preprocessor limitation): `YADE_PLUGIN((classFoo));`. If there is no class in this file, do not use this macro at all.

Internally, this macro creates function `registerThisPluginClasses_` declared specially as `__attribute__((constructor))` (see GCC Function Attributes); this attributes makes the function being executed when the plugin is loaded via `dlopen` from `ClassFactory::load(...)`. It registers all factorable classes from that file in the *Class factory*.

Note

Classes that do not derive from `Factorable`, such as `Shop` or `SpherePack`, are not declared with `YADE_PLUGIN`.

This is an example of a serializable class header:

```
namespace yade {
/*! Homogeneous gravity field; applies gravity×mass force on all bodies. */
class GravityEngine: public GlobalEngine{
        public:
                virtual void action();
        // registering class and its base for the RTTI system
        YADE_CLASS_BASE_DOC_ATTRS(GravityEngine,GlobalEngine,
                // documentation visible from python and generated reference␣
↪documentation
                "Homogeneous gravity field; applies gravity×mass force on all bodies.
↪",
                // enumerating attributes here, include documentation
                ((Vector3r,gravity,Vector3r::Zero(),"acceleration, zero by default␣
↪[kgms $^{2}$]"))
        );
};
// registration function for ClassFactory
REGISTER_SERIALIZABLE(GravityEngine);
} // namespace yade
```

and this is the implementation:

```
#include <pkg/common/GravityEngine.hpp>
#include <core/Scene.hpp>

namespace yade {
// registering the plugin
YADE_PLUGIN((GravityEngine));

void GravityEngine::action(){
        /* do the work here */
}
} // namespace yade
```

We can create a mini-simulation (with only one GravityEngine):

```
Yade [18]: O.engines=[GravityEngine(gravity=Vector3(0,0,-9.81))]

Yade [19]: O.save('abc.xml')
```

and the XML save looks like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive" version="20">
<scene class_id="0" tracking_level="0" version="1">
        <px class_id="1" tracking_level="1" version="0" object_id="_0">
                <Serializable class_id="2" tracking_level="1" version="0" object_id="_
↪1"></Serializable>
                <dt>1.0000000000000002e-08</dt>
                <iter>0</iter>
                <subStepping>0</subStepping>
                <subStep>-1</subStep>
                <time>0.0000000000000000e+00</time>
                <speed>0.0000000000000000e+00</speed>
                <stopAtIter>0</stopAtIter>
                <stopAtTime>0.0000000000000000e+00</stopAtTime>
                <isPeriodic>0</isPeriodic>
                <trackEnergy>0</trackEnergy>
                <doSort>0</doSort>
                <runInternalConsistencyChecks>1</runInternalConsistencyChecks>
                <selectedBody>-1</selectedBody>
                <subdomain>0</subdomain>
                <subD class_id="3" tracking_level="0" version="1">
                        <px class_id="4" tracking_level="1" version="0" object_id="_
↪2">
                                <Serializable object_id="_3"></Serializable>
                                <color class_id="5" tracking_level="0" version="0">
                                        <x>1.0000000000000000e+00</x>
                                        <y>1.0000000000000000e+00</y>
                                        <z>1.0000000000000000e+00</z>
                                </color>
                                <wire>0</wire>
                                <highlight>0</highlight>
                        </px>
                </subD>
                <tags class_id="6" tracking_level="0" version="0">
                        <count>5</count>
                        <item_version>0</item_version>
```

(continues on next page)

(continued from previous page)

```
                        <item>author=root~(root@yade-ci)</item>
                        <item>isoTime=20261020T045757</item>
                        <item>id=20261020T045757p3349</item>
                        <item>d.id=20261020T045757p3349</item>
                        <item>id.d=20261020T045757p3349</item>
                </tags>
                <engines class_id="7" tracking_level="0" version="0">
                        <count>1</count>
                        <item_version>1</item_version>
                        <item class_id="8" tracking_level="0" version="1">
                                <px class_id="10" class_name="yade::GravityEngine"␣
↪tracking_level="1" version="0" object_id="_4">
                                        <FieldApplier class_id="11" tracking_level=
↪"1" version="0" object_id="_5">
                                                <GlobalEngine class_id="12"␣
↪tracking_level="1" version="0" object_id="_6">
                                                        <Engine class_id="9"␣
↪tracking_level="1" version="0" object_id="_7">
                                                                <Serializable␣
↪object_id="_8"></Serializable>
                                                                <dead>0</dead>
                                                                <ompThreads>-1</
↪ompThreads>
                                                                <label></label>
                                                        </Engine>
                                                </GlobalEngine>
                                        </FieldApplier>
                                        <gravity>
                                                <x>0.00000000000000000e+00</x>
                                                <y>0.00000000000000000e+00</y>
                                                <z>-9.81000000000000050e+00</z>
                                        </gravity>
                                        <mask>0</mask>
                                        <warnOnce>1</warnOnce>
                                </px>
                        </item>
                </engines>
                <_nextEngines>
                        <count>0</count>
                        <item_version>1</item_version>
                </_nextEngines>
                <bodies class_id="13" tracking_level="0" version="1">
                        <px class_id="14" tracking_level="1" version="0" object_id="_
↪9">
                                <Serializable object_id="_10"></Serializable>
                                <body class_id="15" tracking_level="0" version="0">
                                        <count>0</count>
                                        <item_version>1</item_version>
                                </body>
                                <insertedBodies>
                                        <count>0</count>
                                        <item_version>0</item_version>
                                </insertedBodies>
                                <erasedBodies>
                                        <count>0</count>
                                        <item_version>0</item_version>
```

(continues on next page)

(continued from previous page)

```
                                </erasedBodies>
                                <realBodies>
                                        <count>0</count>
                                        <item_version>0</item_version>
                                </realBodies>
                                <useRedirection>0</useRedirection>
                                <enableRedirection>1</enableRedirection>
                                <subdomainBodies>
                                        <count>0</count>
                                        <item_version>0</item_version>
                                </subdomainBodies>
                        </px>
                </bodies>
                <interactions class_id="17" tracking_level="0" version="1">
                        <px class_id="18" tracking_level="1" version="0" object_id="_
↪11">
                                <Serializable object_id="_12"></Serializable>
                                <interaction class_id="19" tracking_level="0" version=
↪"0">
                                        <count>0</count>
                                        <item_version>1</item_version>
                                </interaction>
                                <serializeSorted>0</serializeSorted>
                                <dirty>1</dirty>
                        </px>
                </interactions>
                <energy class_id="20" tracking_level="0" version="1">
                        <px class_id="21" tracking_level="1" version="0" object_id="_
↪13">
                                <Serializable object_id="_14"></Serializable>
                                <energies class_id="22" tracking_level="0" version=
↪"0">
                                        <size>0</size>
                                </energies>
                                <names class_id="23" tracking_level="0" version="0">
                                        <count>0</count>
                                        <item_version>0</item_version>
                                </names>
                                <resetStep>
                                        <count>0</count>
                                </resetStep>
                        </px>
                </energy>
                <materials class_id="25" tracking_level="0" version="0">
                        <count>0</count>
                        <item_version>1</item_version>
                </materials>
                <bound class_id="26" tracking_level="0" version="1">
                        <px class_id="-1"></px>
                </bound>
                <cell class_id="28" tracking_level="0" version="1">
                        <px class_id="29" tracking_level="1" version="0" object_id="_
↪15">
                                <Serializable object_id="_16"></Serializable>
                                <trsf class_id="30" tracking_level="0" version="0">
                                        <m00>1.0000000000000000e+00</m00>
```

(continues on next page)

(continued from previous page)

```
                        <m01>0.0000000000000000e+00</m01>
                        <m02>0.0000000000000000e+00</m02>
                        <m10>0.0000000000000000e+00</m10>
                        <m11>1.0000000000000000e+00</m11>
                        <m12>0.0000000000000000e+00</m12>
                        <m20>0.0000000000000000e+00</m20>
                        <m21>0.0000000000000000e+00</m21>
                        <m22>1.0000000000000000e+00</m22>
                </trsf>
                <refHSize>
                        <m00>1.0000000000000000e+00</m00>
                        <m01>0.0000000000000000e+00</m01>
                        <m02>0.0000000000000000e+00</m02>
                        <m10>0.0000000000000000e+00</m10>
                        <m11>1.0000000000000000e+00</m11>
                        <m12>0.0000000000000000e+00</m12>
                        <m20>0.0000000000000000e+00</m20>
                        <m21>0.0000000000000000e+00</m21>
                        <m22>1.0000000000000000e+00</m22>
                </refHSize>
                <hSize>
                        <m00>1.0000000000000000e+00</m00>
                        <m01>0.0000000000000000e+00</m01>
                        <m02>0.0000000000000000e+00</m02>
                        <m10>0.0000000000000000e+00</m10>
                        <m11>1.0000000000000000e+00</m11>
                        <m12>0.0000000000000000e+00</m12>
                        <m20>0.0000000000000000e+00</m20>
                        <m21>0.0000000000000000e+00</m21>
                        <m22>1.0000000000000000e+00</m22>
                </hSize>
                <prevHSize>
                        <m00>1.0000000000000000e+00</m00>
                        <m01>0.0000000000000000e+00</m01>
                        <m02>0.0000000000000000e+00</m02>
                        <m10>0.0000000000000000e+00</m10>
                        <m11>1.0000000000000000e+00</m11>
                        <m12>0.0000000000000000e+00</m12>
                        <m20>0.0000000000000000e+00</m20>
                        <m21>0.0000000000000000e+00</m21>
                        <m22>1.0000000000000000e+00</m22>
                </prevHSize>
                <velGrad>
                        <m00>0.0000000000000000e+00</m00>
                        <m01>0.0000000000000000e+00</m01>
                        <m02>0.0000000000000000e+00</m02>
                        <m10>0.0000000000000000e+00</m10>
                        <m11>0.0000000000000000e+00</m11>
                        <m12>0.0000000000000000e+00</m12>
                        <m20>0.0000000000000000e+00</m20>
                        <m21>0.0000000000000000e+00</m21>
                        <m22>0.0000000000000000e+00</m22>
                </velGrad>
                <nextVelGrad>
                        <m00>0.0000000000000000e+00</m00>
                        <m01>0.0000000000000000e+00</m01>
```

(continues on next page)

(continued from previous page)

```
                                        <m02>0.0000000000000000e+00</m02>
                                        <m10>0.0000000000000000e+00</m10>
                                        <m11>0.0000000000000000e+00</m11>
                                        <m12>0.0000000000000000e+00</m12>
                                        <m20>0.0000000000000000e+00</m20>
                                        <m21>0.0000000000000000e+00</m21>
                                        <m22>0.0000000000000000e+00</m22>
                                </nextVelGrad>
                                <prevVelGrad>
                                        <m00>0.0000000000000000e+00</m00>
                                        <m01>0.0000000000000000e+00</m01>
                                        <m02>0.0000000000000000e+00</m02>
                                        <m10>0.0000000000000000e+00</m10>
                                        <m11>0.0000000000000000e+00</m11>
                                        <m12>0.0000000000000000e+00</m12>
                                        <m20>0.0000000000000000e+00</m20>
                                        <m21>0.0000000000000000e+00</m21>
                                        <m22>0.0000000000000000e+00</m22>
                                </prevVelGrad>
                                <homoDeform>2</homoDeform>
                                <velGradChanged>0</velGradChanged>
                                <flipFlippable>0</flipFlippable>
                        </px>
                </cell>
                <miscParams class_id="31" tracking_level="0" version="0">
                        <count>0</count>
                        <item_version>1</item_version>
                </miscParams>
                <dispParams class_id="32" tracking_level="0" version="0">
                        <count>0</count>
                        <item_version>1</item_version>
                </dispParams>
        </px>
</scene>
</boost_serialization>
```

Warning

Since XML files closely reflect implementation details of Yade, they will not be compatible between different versions. Use them only for short-term saving of scenes. Python is *the* high-level description Yade uses.

Python attribute access

The macro *YADE_CLASS_BASE_DOC_* macro family* introduced above is (behind the scenes) also used to create functions for accessing attributes from Python. As already noted, set of serialized attributes and set of attributes accessible from Python are identical. Besides attribute access, these wrapper classes imitate also some functionality of regular python dictionaries:

```
Yade [20]: s=Sphere()

Yade [21]: s.radius                ## read-access
Out[21]: nan

Yade [22]: s.radius=4.             ## write access
```

(continues on next page)

(continued from previous page)

```
Yade [23]: s.dict().keys()                ## show all available keys
Out[23]: dict_keys(['radius', 'color', 'wire', 'highlight'])

Yade [24]: for k in s.dict().keys(): print(s.dict()[k])  ## iterate over keys, print␣
↪their values
   ....:
4.0
Vector3(1,1,1)
False
False

Yade [25]: s.dict()['radius']              ## same as: 'radius' in s.keys()
Out[25]: 4.0

Yade [26]: s.dict()                        ## show dictionary of both attributes and␣
↪values
Out[26]: {'radius': 4.0, 'color': Vector3(1,1,1), 'wire': False, 'highlight': False}
```

YADE_CLASS_BASE_DOC_* macro family

There are several macros that hide behind them the functionality of *Sphinx documentation*, *Run-time type identification (RTTI)*, *Attribute registration*, *Python attribute access*, plus automatic attribute initialization and documentation. They are all defined as shorthands for the macro `YADE_CLASS_BASE_DOC_ATTRS_INIT_CTOR_PY`, which is itself a shorthand for the base macro `YADE_CLASS_BASE_DOC_ATTRS_DEPREC_INIT_CTOR_PY` with some arguments left out. They must be placed in class declaration's body (`.hpp` file):

```
#define YADE_CLASS_BASE_DOC(klass,base,doc) \
        YADE_CLASS_BASE_DOC_ATTRS(klass,base,doc,)
#define YADE_CLASS_BASE_DOC_ATTRS(klass,base,doc,attrs) \
        YADE_CLASS_BASE_DOC_ATTRS_CTOR(klass,base,doc,attrs,)
#define YADE_CLASS_BASE_DOC_ATTRS_CTOR(klass,base,doc,attrs,ctor) \
        YADE_CLASS_BASE_DOC_ATTRS_CTOR_PY(klass,base,doc,attrs,ctor,)
#define YADE_CLASS_BASE_DOC_ATTRS_CTOR_PY(klass,base,doc,attrs,ctor,py) \
        YADE_CLASS_BASE_DOC_ATTRS_INIT_CTOR_PY(klass,base,doc,attrs,,ctor,py)
#define YADE_CLASS_BASE_DOC_ATTRS_INIT_CTOR_PY(klass,base,doc,attrs,init,ctor,py) \
        YADE_CLASS_BASE_DOC_ATTRS_INIT_CTOR_PY(klass,base,doc,attrs,inits,ctor,py)
```

Expected parameters are indicated by macro name components separated with underscores. Their meaning is as follows:

klass
: (unquoted) name of this class (used for RTTI and python)

base
: (unquoted) name of the base class (used for RTTI and python)

doc
: docstring of this class, written in the ReST syntax. This docstring will appear in generated documentation (such as *CpmMat*). It can be as long as necessary, use string literal to avoid sequences interpreted by c++ compiler (so that some backslashes don't have to be doubled, like in $\sigma = \varepsilon E$) instead of writing this:

  ```
  ":math:`\\sigma=\\epsilon E"
  ```

  Write following: `R"""(:math:`\sigma=\epsilon E`)"""`. When the `R"""(raw text)"""` is used the escaped characters `\n` and `\t` do not have to be written. Newlines and tabs can be used instead. For example see *here* and here. Hyperlink the documentation abundantly with `yref` (all references

to other classes should be hyperlinks). See *previous section* about syntax on using references and anchors.

attrs

Attribute must be written in the form of parethesized list:

```
((type1,attr1,initValue1,attrFlags,"Attribute 1 documentation"))
((type2,attr2,,,"Attribute 2 documentation"))  // initValue and attrFlags␣
↪unspecified
```

This will expand to

1. data members declaration in c++ (note that all attributes are *public*):

   ```
   public: type1 attr1;
           type2 attr2;
   ```

2. Initializers of the default (argument-less) constructor, for attributes that have non-empty `initValue`:

   ```
   Klass(): attr1(initValue1), attr2() { /* constructor body */ }
   ```

   No initial value will be assigned for attribute of which initial value is left empty (as is for attr2 in the above example). Note that you still have to write the commas.

3. Registration of the attribute in the serialization system (unless disabled by attrFlags – *see below*)

4. **Registration of the attribute in python (unless disabled by attrFlags), so that it can be accessed as `klass().name1`.**
   The attribute is read-write by default, see attrFlags to change that.

   This attribute will carry the docstring provided, along with knowledge of the initial value. You can add text description to the default value using the comma operator of c++ and casting the char* to (void):

   ```
   ((Real,dmgTau,((void)"deactivated if negative",-1),,"Characteristic time␣
   ↪for normal viscosity. [s]"))
   ```

   leading to *CpmMat::dmgTau*.

   The attribute is registered via `boost::python::add_property` specifying `return_by_value` policy rather than `return_internal_reference`, which is the default when using `def_readwrite`. The reason is that we need to honor custom converters for those values; see note in *Custom converters* for details.

Attribute flags

By default, an attribute will be serialized and will be read-write from python. There is a number of flags that can be passed as the 4th argument (empty by default) to change that:

- `Attr::noSave` avoids serialization of the attribute (while still keeping its accessibility from Python)
- `Attr::readonly` makes the attribute read-only from Python
- `Attr::triggerPostLoad` will trigger call to `postLoad` function to handle attribute change after its value is set from Python; this is to ensure consistency of other precomputed data which depend on this value (such as `Cell.trsf` and such)
- `Attr::hidden` will not expose the attribute to Python at all

- `Attr::noResize` will not permit changing size of the array from Python [not yet used]

Flags can be combined as usual using bitwise disjunction `|` (such as `Attr::noSave | Attr::readonly`), though in such case the value should be parenthesized to avoid a warning with some compilers (g++ specifically), i.e. `(Attr::noSave | Attr::readonly)`.

Currently, the flags logic handled at runtime; that means that even for attributes with `Attr::noSave`, their serialization template must be defined (although it will never be used). In the future, the implementation might be template-based, avoiding this necessity.

deprec

List of deprecated attribute names. The syntax is

```
((oldName1,newName1,"Explanation why renamed etc."))
((oldName2,newName2,"! Explanation why removed and what to do instead."))
```

This will make accessing `oldName1` attribute *from Python* return value of `newName`, but displaying warning message about the attribute name change, displaying provided explanation. This happens whether the access is read or write.

If the explanation's first character is `!` (*bang*), the message will be displayed upon attribute access, but exception will be thrown immediately. Use this in cases where attribute is no longer meaningful or was not straightforwardly replaced by another, but more complex adaptation of user's script is needed. You still have to give `newName2`, although its value will never be used – you can use any variable you like, but something must be given for syntax reasons).

Warning

Due to compiler limitations, this feature only works if Yade is compiled with gcc >= 4.4. In the contrary case, deprecated attribute functionality is disabled, even if such attributes are declared.

init

Parethesized list of the form:

```
((attr3,value3)) ((attr4,value4))
```

which will be expanded to initializers in the default ctor:

```
Klass(): /* attributes declared with the attrs argument */ attr4(value4),␣
↪attr5(value5) { /* constructor body */ }
```

The purpose of this argument is to make it possible to initialize constants and references (which are not declared as attributes using this macro themselves, but separately), as that cannot be done in constructor body. This argument is rarely used, though.

ctor

will be put directly into the generated constructor's body. Mostly used for calling createIndex(); in the constructor.

Note

The code must not contain commas outside parentheses (since preprocessor uses commas to separate macro arguments). If you need complex things at construction time, create a separate init() function and call it from the constructor instead.

`py`
: will be appended directly after generated python code that registers the class and all its attributes. You can use it to access class methods from python, for instance, to override an existing attribute with the same name etc:

```
.def_readonly("omega",&CpmPhys::omega,"Damage internal variable")
.def_readonly("Fn",&CpmPhys::Fn,"Magnitude of normal force.")
```

`def_readonly` will not work for custom types (such as std::vector), as it bypasses conversion registry; see *Custom converters* for details.

Exposing function-attributes to GUI

Usually to expose a more complex data a getter and setter functions are used, for example *Body::mask*. They are accessible from python. To make them visible in GUI without a corresponding variable at all a function `virtual ::boost::python::dict pyDictCustom() const { …… };` must be overridden. For example see Interaction.hpp where a special attribute `isReal` is exposed to GUI. To mark such attribute as readonly an extra information has to be added to its docstring: `:yattrflags:`2``. Normally it is put there by the *class attribute registration macros*. But since it is not a variable, such attribute has to be added manually.

Special python constructors

The Python wrapper automatically creates constructor that takes keyword (named) arguments corresponding to instance attributes; those attributes are set to values provided in the constructor. In some cases, more flexibility is desired (such as *InteractionLoop*, which takes 3 lists of functors). For such cases, you can override the function `Serializable::pyHandleCustomCtorArgs`, which can arbitrarily modify the new (already existing) instance. It should modify in-place arguments given to it, as they will be passed further down to the routine which sets attribute values. In such cases, you should document the constructor:

```
.. admonition:: Special constructor

        Constructs from lists of …
```

which then appears in the documentation similar to *InteractionLoop*.

Enums

It is possible to expose `enum class` in GUI in a dropdown menu. This approach is backward compatible, an assignment of `int` value in an old python script will work the same as before. Additionally it will be possible to assign the `string` type values to an enum. To enable the dropdown menu one must `#include <lib/serialization/EnumSupport.hpp>` and put a macro `YADE_ENUM( Scope , EnumName , (ValueName1)(ValueName2)(ValueName3)(ValueName4) )` in a `.cpp` file. Where each macro argument means:

1. `Scope` is the full scope name in which the enum resides. For example the scope of `yade::OpenGLRenderer::BlinkHighlight` is `yade::OpenGLRenderer`.
2. `EnumName` is the name of the enum type (not variable name!) to be registered
3. `ValueName` are all enum values that are to be exposed to python. They have to be updated if the C++ enum declaration in `.hpp` file changes.

After it is registered, like for example in OpenGLRenderer.cpp it is available for use. Additionally the registered enum class type definitions are exposed in `yade.EnumClass_*` scope, for example one can check the `names` and `values` dictionaries:

```
Yade [27]: yade.EnumClass_BlinkHighlight.names
Out[27]:
```

(continues on next page)

(continued from previous page)

```
{'NEVER': yade.EnumClass_BlinkHighlight.NEVER,
 'NORMAL': yade.EnumClass_BlinkHighlight.NORMAL,
 'WEAK': yade.EnumClass_BlinkHighlight.WEAK}

Yade [28]: yade.EnumClass_BlinkHighlight.values
Out[28]:
{0: yade.EnumClass_BlinkHighlight.NEVER,
 1: yade.EnumClass_BlinkHighlight.NORMAL,
 2: yade.EnumClass_BlinkHighlight.WEAK}
```

Keep in mind that these are **not the variable instances** hence trying to assign something to them will not change the blinkHighlight setting in GUI. To change enum value from python the respective variable must be assigned to, such as *yade.qt.Renderer().blinkHighlight*. Trying to assign an incorrect value will throw an exception. For example:

```
Yade [29]: r = yade.NewtonIntegrator() # this is only a test of enum, not of␣
↪NewtonIntegrator

Yade [30]: r.rotAlgorithm # check current rotation algorithm (also available in the␣
↪GUI Inspector of Engines)
Out[30]: yade.EnumClass_RotAlgorithm.delValle2023

Yade [31]: r.rotAlgorithm = 'Omelyan1998'

Yade [32]: try:
   ....:     r.rotAlgorithm = 20    # assigning incorrect value throws an exception
   ....: except:
   ....:     print("Error, value is still equal to:",r.rotAlgorithm)
   ....:
Error, value is still equal to: Omelyan1998

Yade [33]: r.rotAlgorithm
Out[33]: yade.EnumClass_RotAlgorithm.Omelyan1998
```

Alternatively the dropdown menu in GUI can be used for the same effect.

Static attributes

Some classes (such as OpenGL functors) are instantiated automatically; since we want their attributes to be persistent throughout the session, they are static. To expose class with static attributes, use the `YADE_CLASS_BASE_DOC_STATICATTRS` macro. Attribute syntax is the same as for `YADE_CLASS_BASE_DOC_ATTRS`:

```
class SomeClass: public BaseClass{
        YADE_CLASS_BASE_DOC_STATICATTRS(SomeClass,BaseClass,"Documentation of␣
↪SomeClass",
                ((Type1,attr1,default1,"doc for attr1"))
                ((Type2,attr2,default2,"doc for attr2"))
        );
};
```

additionally, you *have* to allocate memory for static data members in the `.cpp` file (otherwise, error about undefined symbol will appear when the plugin is loaded):

There is no way to expose class that has both static and non-static attributes using `YADE_CLASS_BASE_*` macros. You have to expose non-static attributes normally and wrap static attributes separately in the `py` parameter.

Returning attribute by value or by reference

When attribute is passed from c++ to python, it can be passed either as

- value: new python object representing the original c++ object is constructed, but not bound to it; changing the python object doesn't modify the c++ object, unless explicitly assigned back to it, where inverse conversion takes place and the c++ object is replaced.
- reference: only reference to the underlying c++ object is given back to python; modifying python object will make the c++ object modified automatically.

The way of passing attributes given to `YADE_CLASS_BASE_DOC_ATTRS` in the `attrs` parameter is determined automatically in the following manner:

- **`Vector3`, `Vector3i`, `Vector2`, `Vector2i`, `Matrix3` and `Quaternion` objects are passed by *reference*. For instance::**
  O.bodies[0].state.pos[0]=1.33

  will assign correct value to `x` component of position, without changing the other ones.
- **Yade classes (all that use `shared_ptr` when declared in python: all classes deriving from *Serializable* declared with `YADE_CLASS_BASE_DOC_*`, and some others) are passed as *references* (technically speaking, they are passed by value of the `shared_ptr`, but by virtue of its sharedness, they appear as references). For instance::**
  O.engines[4].damping=.3

  will change *damping* parameter on the original engine object, not on its copy.
- **All other types are passed by *value*. This includes, most importantly, sequence types declared in *Custom converters*, such as `std::vector<shared_ptr<Engine> >`. For this reason, ::**
  O.engines[4]=NewtonIntegrator()

  will *not* work as expected; it will replace 5th element of a *copy* of the sequence, and this change will not propagate back to c++.

Multiple dispatch

Multiple dispatch is generalization of virtual methods: a *Dispatcher* decides based on type(s) of its argument(s) which of its *Functors* to call. Number of arguments (currently 1 or 2) determines *arity* of the dispatcher (and of the functor): unary or binary. For example:

```
InsertionSortCollider([Bo1_Sphere_Aabb(),Bo1_Facet_Aabb()])
```

creates *InsertionSortCollider*, which internally contains *Collider.boundDispatcher*, a *BoundDispatcher* (a *Dispatcher*), with 2 functors; they receive `Sphere` or `Facet` instances and create `Aabb`. This code would look like this in c++:

```
shared_ptr<InsertionSortCollider> collider=(new InsertionSortCollider);
collider->boundDispatcher->add(new Bo1_Sphere_Aabb());
collider->boundDispatcher->add(new Bo1_Facet_Aabb());
```

There are currenly 4 predefined dispatchers (see *dispatcher-names*) and corresponding functor types. They are inherited from template instantiations of `Dispatcher1D` or `Dispatcher2D` (for functors, `Functor1D` or `Functor2D`). These templates themselves derive from `DynlibDispatcher` (for dispatchers) and `FunctorWrapper` (for functors).

Example: IGeomDispatcher

Let's take (the most complicated perhaps) *IGeomDispatcher*. *IGeomFunctor*, which is dispatched based on types of 2 *Shape* instances (a *Functor*), takes a number of arguments and returns bool. The functor "call" is always provided by its overridden `Functor::go` method; it always receives the dispatched instances as first argument(s) (2 × `const shared_ptr<Shape>&`) and a number of other arguments it needs:

```
class IGeomFunctor: public Functor2D<
   bool,                                  //return type
   TYPELIST_7(const shared_ptr<Shape>&,   // 1st class for dispatch
      const shared_ptr<Shape>&,           // 2nd class for dispatch
      const State&,                       // other arguments passed to ::go
      const State&,                       // …
      const Vector3r&,                    // …
      const bool&,                        // …
      const shared_ptr<Interaction>&      // …
   )
>
```

The dispatcher is declared as follows:

```
class IGeomDispatcher: public Dispatcher2D<
   Shape,                        // 1st class for dispatch
   Shape,                        // 2nd class for dispatch
   IGeomFunctor,  // functor type
   bool,                         // return type of the functor

   // follow argument types for functor call
   // they must be exactly the same as types
   // given to the IGeomFunctor above.
   TYPELIST_7(const shared_ptr<Shape>&,
      const shared_ptr<Shape>&,
      const State&,
      const State&,
      const Vector3r&,
      const bool &,
      const shared_ptr<Interaction>&
   ),

   // handle symetry automatically
   // (if the dispatcher receives Sphere+Facet,
   // the dispatcher might call functor for Facet+Sphere,
   // reversing the arguments)
   false
>
{ /* … */ }
```

Functor derived from IGeomFunctor must then

- override the ::go method with appropriate arguments (they must match exactly types given to `TYPELIST_*` macro);
- declare what types they should be dispatched for, and in what order if they are not the same.

```
class Ig2_Facet_Sphere_ScGeom: public IGeomFunctor{
   public:

   // override the IGeomFunctor::go
   //   (it is really inherited from FunctorWrapper template,
   //    therefore not declare explicitly in the
   //    IGeomFunctor declaration as such)
   // since dispatcher dispatches only for declared types
   //   (or types derived from them), we can do
   //   static_cast<Facet>(shape1) and static_cast<Sphere>(shape2)
   //   in the ::go body, without worrying about types being wrong.
```

(continues on next page)

(continued from previous page)

```
   virtual bool go(
      // objects for dispatch
      const shared_ptr<Shape>& shape1, const shared_ptr<Shape>& shape2,
      // other arguments
      const State& state1, const State& state2, const Vector3r& shift2,
      const bool& force, const shared_ptr<Interaction>& c
   );
   /* … */

   // this declares the type we want to be dispatched for, matching
   //   first 2 arguments to ::go and first 2 classes in TYPELIST_7 above
   //   shape1 is a Facet and shape2 is a Sphere
   //   (or vice versa, see lines below)
   FUNCTOR2D(Facet,Sphere);

   // declare how to swap the arguments
   //   so that we can receive those as well
   DEFINE_FUNCTOR_ORDER_2D(Facet,Sphere);
   /* … */
};
```

Dispatch resolution

The dispatcher doesn't always have functors that exactly match the actual types it receives. In the same way as virtual methods, it tries to find the closest match in such way that:

1. the actual instances are derived types of those the functor accepts, or exactly the accepted types;
2. sum of distances from actual to accepted types is sharp-minimized (each step up in the class hierarchy counts as 1)

If no functor is able to accept given types (first condition violated) or multiple functors have the same distance (in condition 2), an exception is thrown.

This resolution mechanism makes it possible, for instance, to have a hierarchy of *ScGeom* classes (for different combination of shapes), but only provide a *LawFunctor* accepting `ScGeom`, rather than having different laws for each shape combination.

Note

Performance implications of dispatch resolution are relatively low. The dispatcher lookup is only done once, and uses fast lookup matrix (1D or 2D); then, the functor found for this type(s) is cached within the `Interaction` (or `Body`) instance. Thus, regular functor call costs the same as dereferencing pointer and calling virtual method. There is blueprint to avoid virtual function call as well.

Note

At the beginning, the dispatch matrix contains just entries exactly matching given functors. Only when necessary (by passing other types), appropriate entries are filled in as well.

Indexing dispatch types

Classes entering the dispatch mechanism must provide for fast identification of themselves and of their parent class.[5] This is called class indexing and all such classes derive from *Indexable*. There are

[5] The functionality described in *Run-time type identification (RTTI)* serves a different purpose (serialization) and would hurt the performance here. For this reason, classes provide numbers (indices) in addition to strings.

`top-level` Indexables (types that the dispatchers accept) and each derived class registers its index related to this top-level Indexable. Currently, there are:

Top-level Indexable	used by
Shape	*BoundFunctor*, *IGeomDispatcher*
Material	*IPhysDispatcher*
IPhys	*LawDispatcher*
IGeom	*LawDispatcher*

The top-level Indexable must use the `REGISTER_INDEX_COUNTER` macro, which sets up the machinery for identifying types of derived classes; they must then use the `REGISTER_CLASS_INDEX` macro *and* call `createIndex()` in their constructor. For instance, taking the *Shape* class (which is a top-level Indexable):

```
// derive from Indexable
class Shape: public Serializable, public Indexable {
   // never call createIndex() in the top-level Indexable ctor!
   /* … */

   // allow index registration for classes deriving from ``Shape``
   REGISTER_INDEX_COUNTER(Shape);
};
```

Now, all derived classes (such as *Sphere* or *Facet*) use this:

```
class Sphere: public Shape{
   /* … */
   YADE_CLASS_BASE_DOC_ATTRS_CTOR(Sphere,Shape,"docstring",
      ((Type1,attr1,default1,"docstring1"))
      /* … */,
      // this is the CTOR argument
         // important; assigns index to the class at runtime
         createIndex();
   );
   // register index for this class, and give name of the immediate parent class
   //    (i.e. if there were a class deriving from Sphere, it would use
   //     REGISTER_CLASS_INDEX(SpecialSphere,Sphere),
   //     not REGISTER_CLASS_INDEX(SpecialSphere,Shape)!)
   REGISTER_CLASS_INDEX(Sphere,Shape);
};
```

At runtime, each class within the top-level Indexable hierarchy has its own unique numerical index. These indices serve to build the dispatch matrix for each dispatcher.

Inspecting dispatch in python

If there is a need to debug/study multiple dispatch, python provides convenient interface for this low-level functionality.

We can inspect indices with the `dispIndex` property (note that the top-level indexable `Shape` has negative (invalid) class index; we purposively didn't call `createIndex` in its constructor):

```
Yade [34]: Sphere().dispIndex, Facet().dispIndex, Wall().dispIndex
Out[34]: (1, 7, 21)

Yade [35]: Shape().dispIndex                    # top-level indexable
Out[35]: -1
```

Dispatch hierarchy for a particular class can be shown with the `dispHierarchy()` function, returning list of class names: 0th element is the instance itself, last element is the top-level indexable (again, with invalid index); for instance:

```
Yade [36]: ScGeom().dispHierarchy()       # parent class of all other ScGeom_ classes
Out[36]: ['ScGeom', 'GenericSpheresContact', 'IGeom']

Yade [37]: ScGridCoGeom().dispHierarchy(), ScGeom6D().dispHierarchy(), CylScGeom().
 ↪dispHierarchy()
Out[37]:
(['ScGridCoGeom', 'ScGeom6D', 'ScGeom', 'GenericSpheresContact', 'IGeom'],
 ['ScGeom6D', 'ScGeom', 'GenericSpheresContact', 'IGeom'],
 ['CylScGeom', 'ScGeom', 'GenericSpheresContact', 'IGeom'])

Yade [38]: CylScGeom().dispHierarchy(names=False)   # show numeric indices instead
Out[38]: [4, 1, 0, -1]
```

Dispatchers can also be inspected, using the .dispMatrix() method:

```
Yade [39]: ig=IGeomDispatcher([
   ....:    Ig2_Sphere_Sphere_ScGeom(),
   ....:    Ig2_Facet_Sphere_ScGeom(),
   ....:    Ig2_Wall_Sphere_ScGeom()
   ....: ])
   ....:

Yade [40]: ig.dispMatrix()
Out[40]:
{('Sphere', 'Sphere'): 'Ig2_Sphere_Sphere_ScGeom',
 ('Sphere', 'Facet'): 'Ig2_Facet_Sphere_ScGeom',
 ('Sphere', 'Wall'): 'Ig2_Wall_Sphere_ScGeom',
 ('Facet', 'Sphere'): 'Ig2_Facet_Sphere_ScGeom',
 ('Wall', 'Sphere'): 'Ig2_Wall_Sphere_ScGeom'}

Yade [41]: ig.dispMatrix(False)          # don't convert to class names
Out[41]:
{(1, 1): 'Ig2_Sphere_Sphere_ScGeom',
 (1, 7): 'Ig2_Facet_Sphere_ScGeom',
 (1, 21): 'Ig2_Wall_Sphere_ScGeom',
 (7, 1): 'Ig2_Facet_Sphere_ScGeom',
 (21, 1): 'Ig2_Wall_Sphere_ScGeom'}
```

We can see that functors make use of symmetry (i.e. that Sphere+Wall are dispatched to the same functor as Wall+Sphere).

Finally, dispatcher can be asked to return functor suitable for given argument(s):

```
Yade [42]: ld=LawDispatcher([Law2_ScGeom_CpmPhys_Cpm()])

Yade [43]: ld.dispMatrix()
Out[43]: {('GenericSpheresContact', 'CpmPhys'): 'Law2_ScGeom_CpmPhys_Cpm'}

# see how the entry for ScGridCoGeom will be filled after this request
Yade [44]: ld.dispFunctor(ScGridCoGeom(),CpmPhys())
Out[44]: <Law2_ScGeom_CpmPhys_Cpm instance at 0x4923c20>

Yade [45]: ld.dispMatrix()
Out[45]:
```

(continues on next page)

(continued from previous page)

```
{('GenericSpheresContact', 'CpmPhys'): 'Law2_ScGeom_CpmPhys_Cpm',
 ('ScGridCoGeom', 'CpmPhys'): 'Law2_ScGeom_CpmPhys_Cpm'}
```

OpenGL functors

OpenGL rendering is being done also by 1D functors (dispatched for the type to be rendered). Since it is sufficient to have exactly one class for each rendered type, the functors are found automatically. Their base functor types are `GlShapeFunctor`, `GlBoundFunctor`, `GlIGeomFunctor` and so on. These classes register the type they render using the `RENDERS` macro:

```
namespace yade { // Cannot have #include directive inside.
class Gl1_Sphere: public GlShapeFunctor {
   public :
      virtual void go(const shared_ptr<Shape>&,
         const shared_ptr<State>&,
         bool wire,
         const GLViewInfo&
      );
   RENDERS(Sphere);
   YADE_CLASS_BASE_DOC_STATICATTRS(Gl1_Sphere,GlShapeFunctor,"docstring",
      ((Type1,staticAttr1,informativeDefault,"docstring"))
      /* … */
   );
};
REGISTER_SERIALIZABLE(Gl1_Sphere);
} // namespace yade
```

You can list available functors of a particular type by querying child classes of the base functor:

```
Yade [46]: yade.system.childClasses('GlShapeFunctor')
Out[46]:
{'Gl1_Box',
 'Gl1_ChainedCylinder',
 'Gl1_Cylinder',
 'Gl1_DeformableElement',
 'Gl1_Facet',
 'Gl1_GridConnection',
 'Gl1_LevelSet',
 'Gl1_Node',
 'Gl1_PFacet',
 'Gl1_Polyhedra',
 'Gl1_PotentialBlock',
 'Gl1_PotentialParticle',
 'Gl1_Sphere',
 'Gl1_Tetra',
 'Gl1_Wall'}
```

Note

OpenGL functors may disappear in the future, being replaced by virtual functions of each class that can be rendered.

Parallel execution

Yade was originally not designed with parallel computation in mind, but rather with maximum flexibility (for good or for bad). Parallel execution was added later; in order to not have to rewrite whole Yade from scratch, relatively non-instrusive way of parallelizing was used: OpenMP. OpenMP is standartized shared-memory parallel execution environment, where parallel sections are marked by special `#pragma` in the code (which means that they can compile with compiler that doesn't support OpenMP) and a few functions to query/manipulate OpenMP runtime if necessary.

There is parallelism at 3 levels:

- Computation, interaction (python, GUI) and rendering threads are separate. This is done via regular threads (boost::threads) and is not related to OpenMP.
- *ParallelEngine* can run multiple engine groups (which are themselves run serially) in parallel; it rarely finds use in regular simulations, but it could be used for example when coupling with an independent expensive computation:

```
ParallelEngine([
        [Engine1(),Engine2()],    # Engine1 will run before Engine2
        [Engine3()]               # Engine3() will run in parallel with␣
↪the group [Engine1(),Engine2()]
                                  # arbitrary number of groups can be used
])
```

`Engine2` will be run after `Engine1`, but in parallel with `Engine3`.

Warning

It is your reponsibility to avoid concurrent access to data when using ParallelEngine. Make sure you understand *very well* what the engines run in parallel do.

- Parallelism inside Engines. Some loops over bodies or interactions are parallelized (notably *InteractionLoop* and *NewtonIntegrator*, which are treated in detail later (FIXME: link)):

```
#pragma omp parallel for
for(long id=0; id<size; id++){
   const shared_ptr<Body>& b(scene->bodies[id]);
   /* … */
}
```

Note

OpenMP requires loops over contiguous range of integers (OpenMP 3 also accepts containers with random-access iterators).

If you consider running parallelized loop in your engine, always evalue its benefits. OpenMP has some overhead fo creating threads and distributing workload, which is proportionally more expensive if the loop body execution is fast. The results are highly hardware-dependent (CPU caches, RAM controller).

Maximum number of OpenMP threads is determined by the `OMP_NUM_THREADS` environment variable and is constant throughout the program run. Yade main program also sets this variable (before loading OpenMP libraries) if you use the `-j/--threads` option. It can be queried at runtime with the `omp_get_max_threads` function.

At places which are susceptible of being accessed concurrently from multiple threads, Yade provides some mutual exclusion mechanisms, discussed elsewhere (FIXME):

- simultaneously writeable container for *ForceContainer*,
- mutex for *Body::state*.

Timing

Yade provides 2 services for measuring time spent in different parts of the code. One has the granularity of engine and can be enabled at runtime. The other one is finer, but requires adjusting and recompiling the code being measured.

Per-engine timing

The coarser timing works by merely accumulating number of invocations and time (with the precision of the `clock_gettime` function) spent in each engine, which can be then post-processed by associated Python module `yade.timing`. There is a static bool variable controlling whether such measurements take place (disabled by default), which you can change

```
TimingInfo::enabled=True;            // in c++
```

```
O.timingEnabled=True                 ## in python
```

After running the simulation, `yade.timing.stats()` function will show table with the results and percentages:

```
Yade [47]: TriaxialTest(numberOfGrains=100).load()

Yade [48]: O.engines[0].label='firstEngine'   ## labeled engines will show by labels␣
↪in the stats table

Yade [49]: import yade.timing;

Yade [50]: O.timingEnabled=True

Yade [51]: yade.timing.reset()                  ## not necessary if used for the␣
↪first time

Yade [52]: O.run(50); O.wait()

Yade [53]: yade.timing.stats()
Name                                                  Count                 Time   ␣
↪        Rel. time
------------------------------------------------------------------------------------
↪-----------------
"firstEngine"                                        50            125.321us         ␣
↪      0.38%
InsertionSortCollider                                26           7566.213us         ␣
↪     23.18%
InteractionLoop                                      50           18081.68us         ␣
↪     55.39%
GlobalStiffnessTimeStepper                            2             56.975us         ␣
↪      0.17%
```

(continues on next page)

(continued from previous page)

```
TriaxialCompressionEngine                          50          2035.267us
↪      6.23%
TriaxialStateRecorder                               3           583.932us
↪      1.79%
NewtonIntegrator                                   50          4197.663us
↪     12.86%
  forces sync                                        50            51.119us
↪        1.22%
  motion integration                                 50          4029.687us
↪       96.00%
  sync max vel                                       50            28.308us
↪        0.67%
  terminate                                          50            18.258us
↪        0.43%
  TOTAL                                             200          4127.372us
↪       98.33%
TOTAL                                                         32647.051us
↪    100.00%
```

Exec count and time can be accessed and manipulated through `Engine::timingInfo` from c++ or `Engine().execCount` and `Engine().execTime` properties in Python.

In-engine and in-functor timing

Timing within engines (and functors) is based on *TimingDeltas* class which is by default instantiated in engines and functors as Engine::timingDeltas and Functor::timingDeltas (*Engine.timingDeltas* and *Functor.timingDeltas* in Python). It is made for timing loops (functors' loop is in their respective dispatcher) and stores cummulatively time differences between *checkpoints*.

Note

Fine timing with `TimingDeltas` will only work if timing is enabled globally (see previous section). The code would still run, but giving zero times and exec counts.

1. Preferably define the timingDeltas attributes in the constructor:

```
// header file
class Law2_ScGeom_CpmPhys_Cpm: public LawFunctor {
   /* … */
   YADE_CLASS_BASE_DOC_ATTRS_CTOR(Law2_ScGeom_CpmPhys_Cpm,LawFunctor,
↪"docstring",
      /* attrs */,
      /* constructor */
      timingDeltas=shared_ptr<TimingDeltas>(new TimingDeltas); //␣
↪timingDeltas object is automatically initialized when using -
↪DENABLE_PROFILING=1 cmake option
   );
   // ...
};
```

2. Inside the loop, start the timing by calling `timingDeltas->start();`
3. At places of interest, call `timingDeltas->checkpoint("label")`. The label is used only for post-processing, data are stored based on the checkpoint position, not the label.

Warning

Checkpoints must be always reached in the same order, otherwise the timing data will be garbage. Your code can still branch, but you have to put checkpoints to places which are in common.

```
void Law2_ScGeom_CpmPhys_Cpm::go(shared_ptr<IGeom>& _geom,
                                 shared_ptr<IPhys>& _phys,
                                 Interaction* I,
                                 Scene* scene)
{
   timingDeltas->start();                        // the point at which␣
↪the first timing starts
   // prepare some variables etc here
   timingDeltas->checkpoint("setup");
   // find geometrical data (deformations) here
   timingDeltas->checkpoint("geom");
   // compute forces here
   timingDeltas->checkpoint("material");
   // apply forces, cleanup here
   timingDeltas->checkpoint("rest");
}
```

4. **Alternatively, you can compile Yade using -DENABLE_PROFILING=1 cmake option and use predefined macros TIMING_DELTAS_START and TIMING_DELTAS_CHECKPOINT. Without -DENABLE_PROFILING options, those macros are empty and do nothing.**

```
void Law2_ScGeom_CpmPhys_Cpm::go(shared_ptr<IGeom>& _geom,
                                 shared_ptr<IPhys>& _phys,
                                 Interaction* I,
                                 Scene* scene)
{
   TIMING_DELTAS_START();
   // prepare some variables etc here
   TIMING_DELTAS_CHECKPOINT("setup")
   // find geometrical data (deformations) here
   TIMING_DELTAS_CHECKPOINT("geom")
   // compute forces here
   TIMING_DELTAS_CHECKPOINT("material")
   // apply forces, cleanup here
   TIMING_DELTAS_CHECKPOINT("rest")
}
```

The output might look like this (note that functors are nested inside dispatchers and `TimingDeltas` inside their engine/functor):

```
Name                                    Count                  Time          Rel. time
-----------------------------------------------------------------------------------------
ForceReseter                          400                9449 s              0.01%
BoundDispatcher                       400             1171770 s              1.15%
InsertionSortCollider                 400             9433093 s              9.24%
IGeomDispatcher        400            15177607 s              14.87%
IPhysDispatcher         400             9518738 s               9.33%
LawDispatcher                         400            64810867 s             63.49%
  Law2_ScGeom_CpmPhys_Cpm
    setup                             4926145             7649131 s              15.25%
```

(continues on next page)

(continued from previous page)

```
    geom                    4926145          23216292 s          46.28%
    material                4926145           8595686 s          17.14%
    rest                    4926145          10700007 s          21.33%
    TOTAL                                    50161117 s         100.00%
NewtonIntegrator                400           1866816 s           1.83%
"strainer"                      400             21589 s           0.02%
"plotDataCollector"             160             64284 s           0.06%
"damageChecker"                   9              3272 s           0.00%
TOTAL                                       102077490 s         100.00%
```

Warning

Do not use *TimingDeltas* in parallel sections, results might not be meaningful. In particular, avoid timing functors inside *InteractionLoop* when running with multiple OpenMP threads.

`TimingDeltas` data are accessible from Python as list of (*label*,*time*,*count*) tuples, one tuple representing each checkpoint:

```
deltas=someEngineOrFunctor.timingDeltas.data()
deltas[0][0] # 0th checkpoint label
deltas[0][1] # 0th checkpoint time in nanoseconds
deltas[0][2] # 0th checkpoint execution count
deltas[1][0] # 1st checkpoint label
             # …
deltas.reset()
```

Timing overhead

The overhead of the coarser, per-engine timing, is very small. For simulations with at least several hundreds of elements, they are below the usual time variance (a few percent).

The finer *TimingDeltas* timing can have major performance impact and should be only used during debugging and performance-tuning phase. The parts that are file-timed will take disproportionally longer time that the rest of engine; in the output presented above, *LawDispatcher* takes almost of total simulation time in average, but the number would be twice of thrice lower typically (note that each checkpoint was timed almost 5 million times in this particular case).

OpenGL Rendering

Yade provides 3d rendering based on QGLViewer. It is not meant to be full-featured rendering and post-processing, but rather a way to quickly check that scene is as intended or that simulation behaves sanely.

Note

Although 3d rendering runs in a separate thread, it has performance impact on the computation itself, since interaction container requires mutual exclusion for interaction creation/deletion. The `InteractionContainer::drawloopmutex` is either held by the renderer (*OpenGLRenderingEngine*) or by the insertion/deletion routine.

Warning

There are 2 possible causes of crash, which are not prevented because of serious performance penalty that would result:

1. access to *BodyContainer*, in particular deleting bodies from simulation; this is a rare operation, though.
2. deleting Interaction::phys or Interaction::geom.

Renderable entities (*Shape*, *State*, *Bound*, *IGeom*, *IPhys*) have their associated *OpenGL functors*. An entity is rendered if

1. Rendering such entities is enabled by appropriate attribute in *OpenGLRenderingEngine*
2. Functor for that particular entity type is found via the *dispatch mechanism*.

`Gl1_*` functors operating on Body's attributes (*Shape*, *State*, *Bound*) are called with the OpenGL context translated and rotated according to *State::pos* and *State::ori*. Interaction functors work in global coordinates.

3.1.7 Simulation framework

Besides the support framework mentioned in the previous section, some functionality pertaining to simulation itself is also provided.

There are special containers for storing bodies, interactions and (generalized) forces. Their internal functioning is normally opaque to the programmer, but should be understood as it can influence performance.

Scene

`Scene` is the object containing the whole simulation. Although multiple scenes can be present in the memory, only one of them is active. Saving and loading (serializing and deserializing) the `Scene` object should make the simulation run from the point where it left off.

Note

All *Engines* and functors have interally a `Scene* scene` pointer which is updated regularly by engine/functor callers; this ensures that the current scene can be accessed from within user code.

For outside functions (such as those called from python, or static functions in `Shop`), you can use `Omega::instance().getScene()` to retrieve a `shared_ptr<Scene>` of the current scene.

Body container

Body container is linear storage of bodies. Each body in the simulation has its unique *id*, under which it must be found in the *BodyContainer*. Body that is not yet part of the simulation typically has id equal to invalid value `Body::ID_NONE`, and will have its `id` assigned upon insertion into the container. The requirements on *BodyContainer* are

- O(1) access to elements,
- linear-addressability (0…n indexability),
- store `shared_ptr`, not objects themselves,
- *no* mutual exclusion for insertion/removal (this must be assured by the called, if desired),
- intelligent allocation of `id` for new bodies (tracking removed bodies),
- easy iteration over all bodies.

Note

Currently, there is "abstract" class `BodyContainer`, from which derive concrete implementations; the initial idea was the ability to select at runtime which implementation to use (to find one that performs the best for given simulation). This incurs the penalty of many virtual function calls, and will probably change in the future. All implementations of BodyContainer were removed in the meantime, except `BodyVector` (internally a `vector<shared_ptr<Body> >` plus a few methods around), which is the fastest.

Insertion/deletion

Body insertion is typically used in *FileGenerator*'s:

```
shared_ptr<Body> body(new Body);
// … (body setup)
scene->bodies->insert(body); // assigns the id
```

Bodies are deleted only rarely:

```
scene->bodies->erase(id);
```

Warning

Since mutual exclusion is not assured, never insert/erase bodies from parallel sections, unless you explicitly assure there will be no concurrent access.

Iteration

The container can be iterated over using `for(const auto& …… : …… )` C++ syntax:

```
for(const auto& b : *scene->bodies){
   if(!b) continue;                        // skip deleted bodies, nullptr-check
   /* do something here */
}
```

The same loop can be also written by using the type `const shared_ptr<Body>&` explicitly:

```
for(const shared_ptr<Body>& b : *scene->bodies){
   if(!b) continue;                        // skip deleted bodies, nullptr-check
   /* do something here */
}
```

Warning

The previously used macro `FOREACH` is now deprecated.

Note a few important things:

1. Always use `const shared_ptr<Body>&` (const reference); that avoids incrementing and decrementing the reference count on each `shared_ptr`.
2. Take care to skip NULL bodies (`if(!b) continue`): deleted bodies are deallocated from the container, but since body id's must be persistent, their place is simply held by an empty `shared_ptr<Body>()` object, which is implicitly convertible to `false`.

In python, the BodyContainer wrapper also has iteration capabilities; for convenience (which is different from the c++ iterator), NULL bodies as silently skipped:

```
Yade [54]: O.bodies.append([Body(),Body(),Body()])
Out[54]: [0, 1, 2]

Yade [55]: O.bodies.erase(1)
Out[55]: True

Yade [56]: [b.id for b in O.bodies]
Out[56]: [0, 2]
```

In loops parallelized using OpenMP, the loop must traverse integer interval (rather than using iterators):

```
const long size=(long)bodies.size();       // store this value, since it doesn't␣
↪change during the loop
#pragma omp parallel for
for(long _id=0; _id<size; _id++){
   const shared_ptr<Body>& b(bodies[_id]);
   if(!b) continue;
   /* … */
}
```

InteractionContainer

Interactions are stored in special container, and each interaction must be uniquely identified by pair of ids (id1,id2).

- O(1) access to elements,
- linear-addressability (0…n indexability),
- store `shared_ptr`, not objects themselves,
- mutual exclusion for insertion/removal,
- easy iteration over all interactions,
- addressing symmetry, i.e. interaction(id1,id2) interaction(id2,id1)

Note

As with BodyContainer, there is "abstract" class InteractionContainer, and then its concrete implementations. Currently, only InteractionVecMap implementation is used and all the other were removed. Therefore, the abstract InteractionContainer class may disappear in the future, to avoid unnecessary virtual calls.

Further, there is a blueprint for storing interactions inside bodies, as that would give extra advantage of quickly getting all interactions of one particular body (currently, this necessitates loop over all interactions); in that case, InteractionContainer would disappear.

Insert/erase

Creating new interactions and deleting them is delicate topic, since many eleents of simulation must be synchronized; the exact workflow is described in *Handling interactions*. You will almost certainly never need to insert/delete an interaction manually from the container; if you do, consider designing your code differently.

```
// both insertion and erase are internally protected by a mutex,
// and can be done from parallel sections safely
```

(continues on next page)

(continued from previous page)

```
scene->interactions->insert(shared_ptr<Interaction>(new Interactions(id1,id2)));
scene->interactions->erase(id1,id2);
```

Iteration

As with BodyContainer, iteration over interactions should use the `for(const auto& …… : …… )` C++ syntax, also `const shared_ptr<Interaction>&` can be used instead of `auto&`:

```
for(const shared_ptr<Interaction>& i : *scene->interactions){
   if(!i->isReal()) continue;
   /* … */
}
```

Warning

The previously used macro `FOREACH` is now deprecated.

Again, note the usage const reference for `i`. The check `if(!i->isReal())` filters away interactions that exist only *potentially*, i.e. there is only *Bound* overlap of the two bodies, but not (yet) overlap of bodies themselves. The `i->isReal()` function is equivalent to `i->geom && i->phys`. Details are again explained in *Handling interactions*.

In some cases, such as OpenMP-loops requiring integral index (OpenMP >= 3.0 allows parallelization using random-access iterator as well), you need to iterate over interaction indices instead:

```
int nIntr=(int)scene->interactions->size(); // hoist container size
#pragma omp parallel for
for(int j=0; j<nIntr; j++){
   const shared_ptr<Interaction>& i=(*scene->interactions)[j];
   if(!i->isReal()) continue;
   /* … */
}
```

ForceContainer

ForceContainer holds "generalized forces", i.e. forces, torques, (explicit) dispalcements and rotations for each body.

During each computation step, there are typically 3 phases pertaining to forces:

1. Resetting forces to zero (usually done by the *ForceResetter* engine)
2. Incrementing forces from parallel sections (solving interactions – from *LawFunctor*)
3. Reading absolute force values sequentially for each body: forces applied from different interactions are summed together to give overall force applied on that body (*NewtonIntegrator*, but also various other engine that read forces)

This scenario leads to special design, which allows fast parallel write access:

- each thread has its own storage (zeroed upon request), and only writes to its own storage; this avoids concurrency issues. Each thread identifies itself by the omp_get_thread_num() function provided by the OpenMP runtime.
- before reading absolute values, the container must be synchronized, i.e. values from all threads are summed up and stored separately. This is a relatively slow operation and we provide ForceContainer::syncCount that you might check to find cummulative number of synchronizations and compare it against number of steps. Ideally, ForceContainer is only synchronized once at each step.

- the container is resized whenever an element outside the current range is read/written to (the read returns zero in that case); this avoids the necessity of tracking number of bodies, but also is potential danger (such as `scene->forces.getForce(1000000000)`, which will probably exhaust your RAM). Unlike c++, Python does check given id against number of bodies.

```
// resetting forces (inside ForceResetter)
scene->forces.reset()

// in a parallel section
scene->forces.addForce(id,force); // add force

// container is not synced after we wrote to it, sync before reading
scene->forces.sync();
const Vector3r& f=scene->forces.getForce(id);
```

Synchronization is handled automatically if values are read from python:

```
Yade [57]: O.bodies.append(Body())
Out[57]: 3

Yade [58]: O.forces.addF(0,Vector3(1,2,3))

Yade [59]: O.forces.f(0)
Out[59]: Vector3(1,2,3)

Yade [60]: O.forces.f(100)
[31m---------------------------------------------------------------------------[39m
[31mIndexError[39m                                Traceback (most recent call last)
[36mCell[39m[36m [39m[32mIn[60][39m[32m, line 1[39m
[32m----> [39m[32m1[39m␣
↪[43mO[49m[43m.[49m[43mforces[49m[43m.[49m[43mf[49m[43m([49m[32;43m100[39;49m[43m)[49m

[31mIndexError[39m: Body id out of range.
```

Handling interactions

Creating and removing interactions is a rather delicate topic and number of components must cooperate so that the whole behaves as expected.

Terminologically, we distinguish

potential interactions,
: having neither *geometry* nor *physics*. *Interaction.isReal* can be used to query the status (`Interaction::isReal()` in c++).

real interactions,
: having both *geometry* and *physics*. Below, we shall discuss the possibility of interactions that only have geometry but no physics.

During each step in the simulation, the following operations are performed on interactions in a typical simulation:

1. Collider creates potential interactions based on spatial proximity. Not all pairs of bodies are susceptible of entering interaction; the decision is done in Collider::mayCollide:
   - clumps may not enter interactions (only their members can)
   - clump members may not interact if they belong to the same clump
   - bitwise AND on both bodies' *masks* must be non-zero (i.e. there must be at least one bit set in common)
2. Collider erases interactions that were requested for being erased (see below).

3. *InteractionLoop* (via *IGeomDispatcher*) calls appropriate *IGeomFunctor* based on *Shape* combination of both bodies, if such functor exists. For real interactions, the functor updates associated *IGeom*. For potential interactions, the functor returns

   `false`
   : if there is no geometrical overlap, and the interaction will stillremain potential-only

   `true`
   : if there is geometrical overlap; the functor will have created an *IGeom* in such case.

   Note

   For *real* interactions, the functor *must* return `true`, even if there is no more spatial overlap between bodies. If you wish to delete an interaction without geometrical overlap, you have to do this in the *LawFunctor*.

   This behavior is deliberate, since different *laws* have different requirements, though ideally using relatively small number of generally useful *geometry functors*.

   Note

   If there is no functor suitable to handle given combination of *shapes*, the interaction will be left in potential state, without raising any error.

4. For real interactions (already existing or just created in last step), *InteractionLoop* (via *IPhysDispatcher*) calls appropriate *IPhysFunctor* based on *Material* combination of both bodies. The functor *must* update (or create, if it doesn't exist yet) associated *IPhys* instance. It is an error if no suitable functor is found, and an exception will be thrown.
5. For real interactions, *InteractionLoop* (via *LawDispatcher*) calls appropriate *LawFunctor* based on combination of *IGeom* and *IPhys* of the interaction. Again, it is an error if no functor capable of handling it is found.
6. *LawDispatcher* takes care of erasing those interactions that are no longer active (such as if bodies get too far apart for non-cohesive laws; or in case of complete damage for damage models). This is triggered by the *LawFunctor* returning false. For this reason it is of upmost importance for the *LawFunctor* to return consistently.

Such interaction will not be deleted immediately, but will be reset to potential state. At the next execution of the collider `InteractionContainer::conditionalyEraseNonReal` will be called, which will completely erase interactions only if the bounding boxes ceased to overlap; the rest will be kept in potential state.

Creating interactions explicitly

Interactions may still be created explicitly with *utils.createInteraction*, without any spatial requirements. This function searches current engines for dispatchers and uses them. *IGeomFunctor* is called with the `force` parameter, obliging it to return `true` even if there is no spatial overlap.

Associating Material and State types

Some models keep extra *state* information in the *Body.state* object, therefore requiring strict association of a *Material* with a certain *State* (for instance, *CpmMat* is associated to *CpmState* and this combination is supposed by engines such as *CpmStateUpdater*).

If a *Material* has such a requirement, it must override 2 virtual methods:

1. *Material.newAssocState*, which returns a new *State* object of the corresponding type. The default implementation returns *State* itself.

2. *Material.stateTypeOk*, which checks whether a given *State* object is of the corresponding type (this check is run at the beginning of the simulation for all particles).

In c++, the code looks like this (for *CpmMat*):

```
class CpmMat: public FrictMat {
   public:
      virtual shared_ptr<State> newAssocState() const { return shared_ptr<State>(new␣
↪CpmState); }
      virtual bool stateTypeOk(State* s) const { return (bool)dynamic_cast<CpmState*>
↪(s); }
   /* ... */
};
```

This allows one to construct *Body* objects from functions such as *utils.sphere* only by knowing the requires *Material* type, enforcing the expectation of the model implementor.

3.1.8 Runtime structure

Startup sequence

Yade's main program is python script in core/main/main.py.in; the build system replaces a few `${variables}` in that file before copying it to its install location. It does the following:

1. Process command-line options, set environment variables based on those options.
2. Import main yade module (`import yade`), residing in py/__init__.py.in. This module locates plugins (recursive search for files `lib*.so` in the `lib` installation directory). yade.boot module is used to setup temporary directory, … and, most importantly, loads plugins.
3. Manage further actions, such as running scripts given at command line, opening *qt.Controller* (if desired), launching the `ipython` prompt.

Singletons

There are several "global variables" that are always accessible from c++ code; properly speaking, they are Singletons, classes of which exactly one instance always exists. The interest is to have some general functionality acessible from anywhere in the code, without the necessity of passing pointers to such objects everywhere. The instance is created at startup and can be always retrieved (as non-const reference) using the `instance()` static method (e.g. `Omega::instance().getScene()`).

There are 3 singletons:

`ClassFactory`
: Registers classes from plugins and able to factor instance of a class given its name as string (the class must derive from `Factorable`). Not exposed to python.

`Omega`
: Access to simulation(s); deserves separate section due to its importance.

`Logging`
: Handles logging filters for all named loggers, see *logging verbosity*.

Omega

The *Omega* class handles all simulation-related functionality: loading/saving, running, pausing.

In python, the wrapper class to the singleton is instantiated[6] as global variable `O`. For convenience, *Omega* is used as proxy for scene's attribute: although multiple `Scene` objects may be instantiated in c++, it is always the current scene that *Omega* represents.

The correspondence of data is literal: *Omega.materials* corresponds to `Scene::materials` of the current scene; likewise for *materials*, *bodies*, *interactions*, *tags*, *cell*, *engines*, *initializers*, *miscParams*.

[6] It is understood that instantiating `Omega()` in python only instantiates the wrapper class, not the singleton itself.

To give an overview of (some) variables:

Python	c++
Omega.iter	`Scene::iter`
Omega.dt	`Scene::dt`
Omega.time	`Scene::time`
Omega.realtime	`Omega::getRealTime()`
Omega.stopAtIter	`Scene::stopAtIter`

`Omega` in c++ contains pointer to the current scene (`Omega::scene`, retrieved by `Omega::instance().getScene()`). Using *Omega.switchScene*, it is possible to swap this pointer with `Omega::sceneAnother`, a completely independent simulation. This can be useful for example (and this motivated this functionality) if while constructing simulation, another simulation has to be run to dynamically generate (i.e. by running simulation) packing of spheres.

Engine loop

Running simulation consists in looping over *Engines* and calling them in sequence. This loop is defined in `Scene::moveToNextTimeStep` function in core/Scene.cpp. Before the loop starts, *O.initializers* are called; they are only run once. The engine loop does the following in each iteration over *O.engines*:

1. set `Engine::scene` pointer to point to the current `Scene`.
2. Call `Engine::isActivated()`; if it returns `false`, the engine is skipped.
3. Call `Engine::action()`
4. If *O.timingEnabled*, increment *Engine::execTime* by the difference from the last time reading (either after the previous engine was run, or immediately before the loop started, if this engine comes first). Increment *Engine::execCount* by 1.

After engines are processed, *virtual time* is incremented by *timestep* and *iteration number* is incremented by 1.

Background execution

The engine loop is (normally) executed in background thread (handled by SimulationFlow class), leaving foreground thread free to manage user interaction or running python script. The background thread is managed by *O.run()* and *O.pause()* commands. Foreground thread can be blocked until the loop finishes using *O.wait()*.

Single iteration can be run without spawning additional thread using *O.step()*.

3.1.9 Python framework

Wrapping c++ classes

Each class deriving from *Serializable* is automatically exposed to python, with access to its (registered) attributes. This is achieved via *YADE_CLASS_BASE_DOC_* macro family*. All classes registered in class factory are default-constructed in `Omega::buildDynlibDatabase`. Then, each serializable class calls `Serializable::pyRegisterClass` virtual method, which injects the class wrapper into (initially empty) `yade.wrapper` module. `pyRegisterClass` is defined by `YADE_CLASS_BASE_DOC` and knows about class, base class, docstring, attributes, which subsequently all appear in boost::python class definition.

Wrapped classes define special constructor taking keyword arguments corresponding to class attributes; therefore, it is the same to write:

```
Yade [61]: f1=ForceEngine()

Yade [62]: f1.ids=[0,4,5]
```

(continues on next page)

(continued from previous page)

```
Yade [63]: f1.force=Vector3(0,-1,-2)
```

and

```
Yade [64]: f2=ForceEngine(ids=[0,4,5],force=Vector3(0,-1,-2))

Yade [65]: print(f1.dict())
{'force': Vector3(0,-1,-2), 'ids': [0, 4, 5], 'dead': False, 'ompThreads': -1,
 ↪'label': ''}

Yade [66]: print(f2.dict())
{'force': Vector3(0,-1,-2), 'ids': [0, 4, 5], 'dead': False, 'ompThreads': -1,
 ↪'label': ''}
```

Wrapped classes also inherit from *Serializable* several special virtual methods: *dict()* returning all registered class attributes as dictionary (shown above), *clone()* returning copy of instance (by copying attribute values), *updateAttrs()* and *updateExistingAttrs()* assigning attributes from given dictionary (the former thrown for unknown attribute, the latter doesn't). And *pyDictCustom()* explained also in *preceeding section*.

Read-only property `name` wraps c++ method `getClassName()` returning class name as string. (Since c++ class and the wrapper class always have the same name, getting python type using `__class__` and its property `__name__` will give the same value).

```
Yade [67]: s=Sphere()

Yade [68]: s.__class__.__name__
Out[68]: 'Sphere'
```

Subclassing c++ types in python

In some (rare) cases, it can be useful to derive new class from wrapped c++ type in pure python. This is done in the *yade.pack module* module: *Predicate* is c++ base class; from this class, several c++ classes are derived (such as *inGtsSurface*), but also python classes (such as the trivial *inSpace* predicate). `inSpace` derives from python class `Predicate`; it is, however, not direct wrapper of the c++ `Predicate` class, since virtual methods would not work.

`boost::python` provides special `boost::python::wrapper` template for such cases, where each overridable virtual method has to be declared explicitly, requesting python override of that method, if present. See Overridable virtual functions for more details.

When python code is called from C++, the calling thread must hold the python "Global Interpreter Lock" (GIL). When initalizing the script as well as running one iteration calling `O.step()`, the running thread is the same as python, and no additional code is required. On the other hand, calling python code inside the simulation loop using `O.run()` needs the lock to be acquired by the thread, or a segfault error will occurs. See implementation of *pyGenericPotential* () for a complete exemple.

Reference counting

Python internally uses reference counting on all its objects, which is not visible to casual user. It has to be handled explicitly if using pure Python/C API with `Py_INCREF` and similar functions.

`boost::python` used in Yade fortunately handles reference counting internally. Additionally, it automatically integrates reference counting for `shared_ptr` and python objects, if class `A` is wrapped as `boost::python::class_<A,shared_ptr<A>>`. Since *all* Yade classes wrapped using *YADE_CLASS_BASE_DOC_* macro family* are wrapped in this way, returning `shared_ptr<…>` objects from is the preferred way of passing objects from c++ to python.

Returning `shared_ptr` is much more efficient, since only one pointer is returned and reference count internally incremented. Modifying the object from python will modify the (same) object in c++ and vice versa. It also makes sure that the c++ object will not be deleted as long as it is used somewhere in python, preventing (important) source of crashes.

Custom converters

When an object is passed from c++ to python or vice versa, then either

1. the type is basic type which is transparently passed between c++ and python (int, bool, std::string etc)
2. the type is wrapped by boost::python (such as Yade classes, `Vector3` and so on), in which case wrapped object is returned;[7]

Other classes, including template containers such as `std::vector` must have their custom converters written separately. Some of them are provided in py/wrapper/customConverters.cpp, notably converters between python (homogeneous, i.e. with all elements of the same type) sequences and c++ `std::vector` of corresponding type; look in that source file to add your own converter or for inspiration.

When an object is crossing c++/python boundary, boost::python's global "converters registry" is searched for class that can perform conversion between corresponding c++ and python types. The "converters registry" is common for the whole program instance: there is no need to register converters in each script (by importing `_customConverters`, for instance), as that is done by yade at startup already.

Note

Custom converters only work for value that are passed by value to python (not "by reference"): some attributes defined using *YADE_CLASS_BASE_DOC_* macro family* are passed by value, but if you define your own, make sure that you read and understand Why is my automatic to-python conversion not being found?.

In short, the default for `def_readwrite` and `def_readonly` is to return references to underlying c++ objects, which avoids performing conversion on them. For that reason, return value policy must be set to `return_by_value` explicitly, using slighly more complicated `add_property` syntax, as explained at the page referenced.

This deficiency is addressed presently in the file lib/serialization/PyClassCustom.hpp for the `.def_readonly(…)` function. It can be improved later if the need arises.

3.1.10 Adding a new python/C++ module

Modules are placed in `py/` directory, the `C++` parts of the modules begin their name with an underscore `_`. The procedure to add a new module is following:

1. Create your new files:
   1. The `yourNewModule.py` file like this.
   2. The `_yourNewModule.cpp` file like this, if part of your module will be written in `C++`.
2. Update the module redirection map in these two places:
   1. `mods` in doc/sphinx/yadeSphinx.py.
   2. `moduleMap` in doc/sphinx/conf.py, if the new module has a `C++` part (this duplication of data will hopefully be soon removed).

[7] Wrapped classes are automatically registered when the class wrapper is created. If wrapped class derives from another wrapped class (and if this dependency is declared with the `boost::python::bases` template, which Yade's classes do automatically), parent class must be registered before derived class, however. (This is handled via loop in `Omega::buildDynlibDatabase`, which reiterates over classes, skipping failures, until they all successfully register) Math classes (Vector3, Matrix3, Quaternion) are wrapped in *minieigenHP*. See *high precision documentation* for more details.

3. Add the C++ file into py/CMakeLists.txt like this.
4. Modify the CMakeLists.txt but only if the file will depend on cmake compilation variables, eg. like this. The file then needs an additional extension .in and be put in two places:
    1. The cmake command to generate the file from .in input: like this.
    2. The cmake command to install it: like this.

Hint

The last step regarding yourNewModule.py.in (or _yourNewModule.cpp.in) is needed only on very rare occasions, and is included here only for the sake of completeness.

Debugging boundary between python and C++

During normal use all C++ exceptions are propagated back to python interface with full information associated with them. The only situation where this might not be the case is during execution of command import module inside a python script. It might happen that when importing a new module some cryptic errors occur like: initialization of module raised unreported exception. These unreported exceptions happen in the situation when the C++ code executed a python code inside it (this is called embedding) and this python code threw an exception. The proper way to deal with this situation is to wrap entire module declaration inside a try {} catch(...) {} block. It might be possible to deal with specific exceptions also (see here for other example catch blocks), however the general solution is to properly inform python that importing this module did not work. In this catch block it is possible to execute PyErr_Print(); command to see what the problem was and propagate the exception back to python, however during import module command only the SystemError python exception can get through. Hence the catch(...) block after BOOST_PYTHON_MODULE(_yourNewModule) should look like this:

```
#include <lib/base/Logging.hpp>

CREATE_CPP_LOCAL_LOGGER("_yourNewModule.cpp");

BOOST_PYTHON_MODULE(_yourNewModule)
try {
        py::def("foo", foo, R"""(
The description of function foo().

:param arg1: description of first argument
:param arg2: description of second argument
:type arg1: type description
:type arg2: type description
:return: return description
:rtype: the return type description

Example usage of foo:

.. ipython::

   In [1]: from yade.yourNewModule import *

   In [1]: foo()

.. note:: Notes, hints and warnings about how to use foo().

        )""");
```

(continues on next page)

(continued from previous page)

```
} catch (...) {
        LOG_FATAL("Importing this module caused an exception and this module is in an␣
↪inconsistent state now.");
        PyErr_Print();
        PyErr_SetString(PyExc_SystemError, __FILE__);
        boost::python::handle_exception();
        throw;
}
```

Note

Pay attention to the `_yourNewModule` inside `BOOST_PYTHON_MODULE(…)`, it has to match the file name of the `.cpp` file.

Further reading, about how to work with python exceptions:

1. Example in boost::python reference manual.
2. Example in boost::python tutorial.
3. When PyErr_Print(); is not enough.

3.1.11 Maintaining compatibility

In Yade development, we identified compatibility to be very strong desire of users. Compatibility concerns python scripts, *not* simulations saved in XML or old c++ code.

Renaming class

Script scripts/rename-class.py should be used to rename class in `c++` code. It takes 2 parameters (old name and new name) and must be run from top-level source directory:

```
$ scripts/rename-class.py OldClassName NewClassName
Replaced 4 occurences, moved 0 files and 0 directories
Update python scripts (if wanted) by running: perl -pi -e 's/\bOldClassName\b/
↪NewClassName/g' `ls **/*.py |grep -v py/system.py`
```

This has the following effects:

1. If file or directory has basename `OldClassName` (plus extension), it will be renamed using `bzr`.
2. All occurences of whole word `OldClassName` will be replaced by `NewClassName` in c++ sources.
3. An entry is added to py/system.py, which contains map of deprecated class names. At yade startup, proxy class with `OldClassName` will be created, which issues a `DeprecationWarning` when being instantiated, informing you of the new name you should use; it creates an instance of `NewClassName`, hence not disruting your script's functioning:

   ```
   Yade [3]: SimpleViscoelasticMat()
   /usr/local/lib/yade-trunk/py/yade/__init__.py:1: DeprecationWarning: Class␣
    ↪`SimpleViscoelasticMat' was renamed to (or replaced by) `ViscElMat', update␣
    ↪your code! (you can run 'yade --update script.py' to do that automatically)
   ->  [3]: <ViscElMat instance at 0x2d06770>
   ```

As you have just been informed, you can run `yade --update` to all old names with their new names in scripts you provide:

```
$ yade-trunk --update script1.py some/where/script2.py
```

This gives you enough freedom to make your class name descriptive and intuitive.

Renaming class attribute

Renaming class attribute is handled from c++ code. You have the choice of merely warning at accessing old attribute (giving the new name), or of throwing exception in addition, both with provided explanation. See `deprec` parameter to *YADE_CLASS_BASE_DOC_* * *macro family* for details.

3.2 Yade on GitLab

3.2.1 Fast checkout (read-only)

Getting the source code without registering on GitLab can be done via a single command. It will not allow interactions with the remote repository, which you access the read-only way:

```
git clone --recurse-submodules https://gitlab.com/yade-dev/trunk.git
```

3.2.2 Branches on GitLab

Most useful commands are listed in the sections below. For more details, see these git guides:

1. ProGit online Book,
2. Guide on setting up git,
3. Git "choose your own adventure",
4. Guide on fixing the conflicts.

Setup

1. Register on gitlab.com
2. Add your SSH key to GitLab
3. Set your username and email through terminal

   ```
   git config --global user.name "Firstname Lastname"
   git config --global user.email "your_email@youremail.com"
   ```

   You can check these settings with `git config --list`.

4. To fork the repository (optional), click the "Fork" button on the gitlab page, and also fork the YADE data files.

   Note

   By default gitlab will try and compile the forked repository, and it will fail if you don't have runners attached to your account. To avoid receiving failure notifications go to repository settings (bottom of left panel->general->permissions) to turn of pipelines.

5. Set Up Your Local Repo through terminal. The argument `--recurse-submodules` is to make sure that `./data` directory is filled with the recent data from yade-data (the path is relative to your gitlab profile):

   ```
   git clone --recurse-submodules git@gitlab.com:username/trunk.git
   ```

   This creates a new folder, named trunk, that contains the whole code (make sure username is replaced by your GitLab name). If you already have a cloned yade repository with `./data` directory in it, then you can populate your existing repository using command:

   ```
   git submodule update --init --recursive
   ```

6. Configure remotes

```
cd to/newly/created/folder
git remote add upstream git@gitlab.com:yade-dev/trunk.git
git remote update
```

Now, your "trunk" folder is linked with two remote repositories both hosted on gitlab.com, the original trunk from yade-dev (called "upstream" after the last command) and the fork which resides in your personal account (called "origin" and always configured by default). Through appropriate commands explained below, you will be able to update your code to include changes commited by others, or to commit yourself changes that others can get.

Holding a fork under personnal account is in fact not strictly necessary. It is recommended, however, and in what follows it is assumed that the above steps have been followed.

Older versions

In case you want to work with, or compile, an older version of Yade which is not tagged, you can create your own (local) branch of the corresponding daily build. Look here for details.

Committing and updating

Inspecting changes

After changing the source code in the local repository you may start by inspecting them with a few commands. For the "diff" command, it is convenient to copy from the output of "status" instead of typing the path to modified files.

```
git status
git diff path/to/modified/file.cpp
```

Pushing changes

Depending on the remote repository you want to push to, follow one of the methods below.

1. Push to yade-dev

   Merging changes into yade-dev's master branch cannot be done directly with a push, only by merge request (see below). It is possible however to push changes to a new branch of yade-dev repository for members of that group. It is currently the only way to have merge requests tested by the gitlab CI pipeline before being effectively merged. To push to a new yade-dev/branch:

```
git branch localBranch
git checkout localBranch
git add path/to/new/file.cpp  #Version a newly created file
git commit path/to/new_or_modified/file.cpp -m 'Commit message'  #stage␣
↪(register) change in the local repository
git pull --rebase upstream master #get updated version of sources from yade-dev␣
↪repo and apply your commits on the top of them
git push upstream localBranch:newlyCreatedBranch #Push all commits to a new␣
↪remote branch.
```

   The first two lines are optional, if ignored the commits will go the to the default branch, called "master". In the last command `localBranch` is the local branch name on which you were working (possibly `master`) and `newlyCreatedBranch` will be the name of that branch on the remote. Please choose a descriptive name as much as you can (e.g. "fixBug457895").

Note

If you run into any problems with command `git pull --rebase upstream master`, you *always can revert* or even better fix the conflicts.

2. Push to personnal repository

   After previous steps proceed to commit through terminal, "localBranch" should be replaced by a relevant name:

```
git branch localBranch
git checkout localBranch
git add path/to/new/file.cpp  #Version a newly created file
git commit path/to/new_or_modified/file.cpp -m 'Commit message'  #stage␣
↪(register) change in the local repository
git push  #Push all commits to the remote branch
```

   The changes will be pushed to your personal fork.

Updating

You may want to get changes done by others to keep your local and remote repositories synced with the upstream:

```
git pull --rebase upstream master #Pull new updates from the upstream to your branch.␣
↪Eq. of "bzr update", updating the local branch from the upstream␣
↪yade-dev/trunk/master
git push  #Merge changes from upstream into your gitlab repo (origin)
```

If you have local uncommited changes this will return an error. A workaround to update while preserving them is to "stash":

```
git stash #backup and hide changes
git pull --rebase upstream master
git push
git stash pop #restore backed up changes
```

Auto rebase

We promote "rebasing" to avoid confusing logs after each commit/pull/push cycle. It can be convenient to setup automatic rebase, so it does not have to be added everytime in the above commands:

```
git config --global branch.autosetuprebase always
```

Now your file `~/.gitconfig` should include:

```
[branch]
  autosetuprebase = always
```

Check also `.git/config` file in your local trunk folder (rebase = true):

```
[remote "origin"]
  url = git@gitlab.com:yade-dev/trunk.git
  fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
  remote = origin
  merge = refs/heads/master
  rebase = true
```

Pulling a rebased branch

If someone else rebased on the gitlab server the branch on which you are working on locally, the command `git pull` may complain that the branches have diverged, and refuse to perform operation, in that case this command:

```
git pull --rebase upstream branchName
```

Will match your local branch history with the one present on the gitlab server.

If you are afraid of messing up your local branch you can always make a copy of this branch with command:

```
git branch backupCopyName
```

If you forgot to make that backup-copy and want to go back, then make a copy anyway and go back with this command:

```
git reset --merge ORIG_HEAD
```

The `ORIG_HEAD` backs up the position of `HEAD` before a potentially dangerous operation (merge, rebase, etc.).

A tutorial on fixing the conflicts is a recommended read.

Note

If you are lost about how to fix your git problems try a git choose your own adventure.

3.2.3 Merge requests

Members of yade-dev

If you have tested your changes and you are ready to merge them into yade-dev's master branch, you'll have to make a "merge request" (MR) from the gitlab.com interface (see the "+" button at the top of the repository webpage). Set source branch and target branch, from yade-dev/trunk/newlyCreatedBranch to yade-dev/trunk/master. The MR will trigger a pipeline which includes compiling, running regression tests, and generating the documentation (the newly built documentation is accessible via settings->pages or by clicking on the "Browse" button in the "Job artifacts" (in the right pane) in the `doc_18_04` build from the pipeline; then navigating to path `Artifacts/install/share/doc`). If the full pipeline succeeds the merge request can be merged into the master branch.

Note

In case of MR to yade-dev's master from another branch of yade-dev, the pipeline will use group runners attached to yade-dev (the group runners are kindly provided by 3SR, UMS Gricad and Gdańsk University of Technology).

New developers

Welcome! At start it is very convenient to work on a local fork of YADE in your own gitlab profile. When you are confident that your changes are ready to be merged into official YADE release, please open a Merge Request (MR) in the following way:

1. Make sure that your work is in a separate branch, not in the `master` branch. You can "copy" your branch into another branch with command `git checkout -b myNewFeature`. Please make sure that the amount of changes as compared to the master branch is not large. In case of larger code improvements it is better to split it into several smaller merge requests. This way it will be faster for us to check it and merge.

2. Push your branch to the repository on your gitlab profile with command such as:

```
git push --set-upstream origin myNewFeature
```

3. You should see something like:

```
remote:
remote: To create a merge request for myNewFeature, visit:
remote:   https://gitlab.com/myProfileName/trunk/-/merge_requests/
↪new?merge_request%5Bsource_branch%5D=myNewFeature
remote:
```

4. When you visit the link mentioned above, you will have to select "Change branches" and make sure that correct target branch is selected. Usually that will be `yade-dev/trunk:master`, because this is the official YADE repository.
5. Fill in the title and description then click "Create merge request" at the bottom of the page.
6. After we review the merge request we can click on it to run in our Continuous Integration (CI) pipeline. This pipeline can't start automatically for security reasons. It will be merged after the pipeline checks pass.

3.2.4 Guidelines for pushing

These are general guidelines for pushing to `yade-dev/trunk`.

1. Set autorebase globaly on the computer (only once see above), or at least on current local branch. Non-rebased pull requests will not be accepted on the upstream. This is to keep history linear, and avoid the merge commits.
2. Inspect the diff to make sure you will not commit junk code (typically some "cout<<" left here and there), using in terminal:

```
git diff file1
```

   Or using your preferred difftool, such as kdiff3:

```
git difftool -t kdiff3 file1
```

   Or, alternatively, any GUI for git: gitg, git-cola…

3. Commit selectively:

```
git commit file1 file2 file3 -m "message" # is good
git commit -a -m "message"                # is bad. It is the best way to␣
↪commit things that should not be commited
```

4. Be sure to work with an up-to-date version launching:

```
git pull --rebase upstream master
```

5. Make sure it compiles and that regression tests pass: try `yade --test` and `yade --check`.

Thanks a lot for your cooperation to Yade!

Chapter 4

Theoretical background and extensions

4.1 DEM formulation

The DEM formulation is presented in earlier chapter 2.1 *DEM formulation* as a common ground for all DEM calculations.

4.2 CFD-DEM coupled simulations with Yade and OpenFOAM

The *FoamCoupling* engine provides a framework for Euler-Lagrange fluid-particle simulation with the open source finite volume solver OpenFOAM. The coupling relies on the Message Passing Interface library (MPI), as OpenFOAM is a parallel solver, furthermore communication between the solvers are realised by MPI messages. The *FoamCoupling* engine must be enabled with the ENABLE_MPI flag during compilation:

```
cmake -DCMAKE_INSTALL_PREFIX=/path/to/install /path/to/source -DENABLE_MPI=1
```

Yade sends the particle information (particle position, velocity, etc.) to all the OpenFOAM processes. Each OpenFOAM process searches the particle in the local mesh, if the particle is found, the hydrodynamic drag force and torque are calculated using the fluid velocity at the particle position (two interpolation methods are available) and the particle velocity. The hydroynamic force is sent to the Yade process and it is added to the force container. The negative of the particle hydrodynamic force (interpolated back to the fluid cell center) is set as source term in the Navier-Stokes equations. Technical details on the coupling methodology can be found in [Kunhappan2017] and [Kunhappan2018].

4.2.1 Supported versions and examples

A number of example scripts can be found in Yade sources, see the OpenFoam folder. Concrete execution can also be seen in the gitlab pipeline, see the test-script.

The supported OpenFoam versions include v10 and v11 from the foundation release, and (not limited to) v2006, v2112, v2212, v2306, v2312 from openfoam.com. The list of versions tested in the development branch can be visualized in gitlab pipelines.

An older version of the coupling, which was using OpenFoam6, is archived in branch FOAM6couplingArchive.

4.2.2 Background

In the standard Euler-Lagrange modelling of particle laden multiphase flows, the particles are treated as point masses. Two approaches are implemented in the present coupling:

1. Point force coupling
2. Volume fraction based force coupling.

In both of the approaches the flow at the particle scale is not resolved and analytical/empirical hydrodynamic force models are used to describe the fluid-particle interactions. For accurate resolution of the particle volume fraction and hydrodynamic forces on the fluid grid the particle size must be smaller than the fluid cell size.

Point force coupling (*icoFoamYade*)

In the point force coupling, the particles are assumed to be smaller than the smallest fluid length scales, such that the particle Reynolds Number is $Re_p < 1.0$. The particle Reynolds number is defined as the ratio of inertial forces to viscous forces. For a sphere, the associated length-scale is the diameter, therefore:

$$Re_p = \frac{\rho_f |\mathbf{U}_r| d_p}{\mu} \tag{4.1}$$

where in (4.1) ρ_f is the fluid density, $|\mathbf{U}_r|$ is the norm of the relative velocity between the particle and the fluid, d_p is the particle diameter and μ the fluid dynamic viscosity. In addition to the Reynolds number, another non-dimensional number that characterizes the particle inertia due to it's mass called Stokes number is defined as:

$$St_k = \frac{\tau_p |\mathbf{U}_f|}{d_p} \tag{4.2}$$

where in equation (4.2) τ_p is the particle relaxation time defined as:

$$\tau_p = \frac{\rho_p d_p^2}{18\mu}$$

For $Re_p < 1$ and $St_k < 1$, the hydrodynamic force on the particle can be represented as a point force. This force is calculated using the Stoke's drag force formulation:

$$F_h = 3\pi\mu d_p (U_f - U_p) \tag{4.3}$$

The force obtained from (4.3) is applied on the particle and in the fluid side (in the cell where the particle resides), this hydrodynamic force is formulated as a body/volume force:

$$f_h = \frac{-F_h}{V_c \rho_f} \tag{4.4}$$

where in equation (4.4) V_c is the volume of the cell and ρ_f is the fluid density. Hence the Navier-Stokes equations for the combined system is:

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot (\mathbf{U}\mathbf{U}) = -\frac{\nabla p}{\rho} + \nabla \bar{\bar{\tau}} + f_h \tag{4.5}$$

Along with the continuity equation:

$$\nabla \cdot \mathbf{U} = 0 \tag{4.6}$$

Volume averaged coupling (*pimpleFoamYade*)

Warning

The volume averaged coupling is currently under active development. Users are advised to exercise caution when utilizing this feature, as some functionalities may be incomplete, experimental, or subject to significant changes in future updates.

In the volume averaged coupling, the effect of the particle volume fraction is included. The Navier-Stokes equations take the following form:

$$\frac{\partial(\varepsilon_f \mathbf{U}_f)}{\partial t} + \nabla \cdot (\varepsilon_f \mathbf{U}_f \mathbf{U}_f) = -\frac{\nabla p}{\rho} + \varepsilon_f \nabla \bar{\bar{\tau}} - K (U_f - U_p) + \mathbf{S}_u + \varepsilon_f \mathbf{g} \tag{4.7}$$

Along with the continuity equation:

$$\frac{\partial \varepsilon_f}{\partial t} + \nabla \cdot (\varepsilon_f \mathbf{U}_f) = 0 \tag{4.8}$$

where in equations (4.7) and (4.8) ε_f is the fluid volume fraction. Note that, we do not solve for ε_f directly, but obtain it from the local particle volume fraction ε_s, $\varepsilon_f = 1 - \varepsilon_s$. K is the particle drag force parameter, $\mathbf{U}_f$ and $\mathbf{U}_p$ are the fluid and particle velocities respectively. $\mathbf{S}_u$ denotes the explicit source term consisting the effect of other hydrodynamic forces such as the Archimedes/ambient force, added mass force etc. Details on the formulation of these forces are presented in the later parts of this section.

The interpolation and averaging of the Eulerean and Lagrangian quantities are based on a Gaussian envelope $G_\star$. In this method, the the effect of the particle is 'seen' by the neighbouring cells of the cell in which it resides. Let $\mathbf{x}_c$ and $\mathbf{x}_p$ be the fluid cell center and particle position respectively, then the Gaussian filter $G_\star(\mathbf{x}_c - \mathbf{x}_p)$ defined as:

$$G_\star(\mathbf{x}_c - \mathbf{x}_p) = \left(2\pi\sigma^2\right)^{\frac{3}{2}} \exp\left(-\frac{\|\mathbf{x}_c - \mathbf{x}_p\|^2}{2\sigma^2}\right) \tag{4.9}$$

with σ being the standard deviation of the filter defined as:

$$\sigma = \delta / \left(2\sqrt{2\ln 2}\right) \tag{4.10}$$

where in equation (4.10) δ is the cut-off range (at present it's set to $3\Delta x$, with Δx being the fluid cell size.) and follows the rule:

$$G_\star(\|\mathbf{x}_c - \mathbf{x}_p\| = \delta/2) = \frac{1}{2} G_\star(\|x_c - x_p\| = 0)$$

The particle volume fraction $\varepsilon_{s,c}$ for a fluid cell c is calculated by:

$$\varepsilon_{s,c} = \frac{\sum_{i=1}^{N_p} V_{p,i} G_{\star(i,c)}}{V_c} \tag{4.11}$$

where in (4.11) N_p is the number of particle contributions on the cell c, $G_{\star(i,c)}$ is the Gaussian weight obtained from (4.9), $V_{p,i} G_{\star(i,c)}$ forms the individual particle volume contribution. V_c is the fluid cell volume and $\varepsilon_f + \varepsilon_s = 1$

The averaging and interpolation of an Eulerean quantity φ from the grid (cells) to the particle position is performed using the following expression:

$$\widetilde{\varphi} = \sum_{i=1}^{N_c} \varphi_i G_{\star(i,p)} \tag{4.12}$$

Hydrodynamic Force

In equation (4.7) the term K is the drag force parameter. In the present implementation, K is based on the Schiller Naumman drag law, which reads as:

$$K = \frac{3}{4} C_d \frac{\rho_f}{d_p} \left|\left| \widetilde{\mathbf{u}}_f - \mathbf{u}_p \right|\right| \varepsilon_f^{-h_{exp}} \tag{4.13}$$

In equation (4.13) ρ_f is the fluid density, d_p the particle diameter, h_{exp} is defined as the 'hindrance coefficient' with the value set as $h_{exp} = 2.65$. The drag force force coefficient C_d is valid for particle Reynolds numbers up to $Re_p < 1000$. The expression for C_d reads as:

$$C_d = \frac{24}{Re_p} \left(1 + 0.15 Re_p^{0.687}\right) \tag{4.14}$$

The expression of hydrodynamic drag force on the particle is:

$$\mathbf{F}_{drag} = V_p K (\widetilde{\mathbf{u}}_f - u_p)$$

In the fluid equations, negative of the drag parameter ($-K$) is distributed back to the grid based on equation (4.11). Since the drag force includes a non-linear dependency on the fluid velocity u_f, this term is set as an implicit source term in the fluid solver.

The Archimedes/ambient force experienced by the particle is calculated as:

$$\mathbf{F}_{by} = \left(\widetilde{-\nabla p} + \widetilde{\nabla \bar{\bar{\tau}}}\right) V_p \tag{4.15}$$

where in (4.15), $\widetilde{\nabla p}$ is the averaged pressure gradient at the particle center and $\widetilde{\nabla \bar{\bar{\tau}}}$ is the averaged divergence of the viscous stress at the particle position.

Added mass force:

$$\mathbf{F}_{am} = C_m \left(\frac{D\widetilde{u}_f}{Dt} - \frac{du_p}{dt} \right) V_p \tag{4.16}$$

where in eqaution (4.16), $\frac{D\widetilde{u}_f}{Dt}$ is the material derivative of the fluid velocity.

Therefore the net hydrodynamic force on the particle reads as:

$$\mathbf{F}_{hyd} = \mathbf{F}_{drag} + \mathbf{F}_{by} + \mathbf{F}_{am}$$

And on the fluid side the explicit source term $\mathbf{S}_{u,c}$ for a fluid cell c is expressed as :

$$\mathbf{S}_{u,c} = \frac{\sum_{i=1}^{N_p} -\mathbf{F}_{hyd,i} \varepsilon_{s,c} G_{\star(i,c)}}{\rho_f V_c}$$

4.2.3 Setting up a case

In Yade

Setting a case in the Yade side is fairly straight forward. The python script describing the scene in Yade is based on this method. Make sure the exact wall/periodic boundary conditions are set in Yade as well as in the OpenFOAM. The particles should not leave the fluid domain. In case a particle has 'escaped' the domain, a warning message would be printed/written to the log file and the simulation will break.

The example in examples/openfoam/scriptYade.py demonstrates the coupling. A symbolic link to Yade is created and it is imported in the script. The MPI environment is initialized by calling the initMPI() function before instantiating the coupling engine

```
initMPI()
fluidCoupling = FoamCoupling()
fluidCoupling.getRank()
```

A list of the particle ids and number of particle is passed to the coupling engine

```
sphereIDs = [b.id for b in O.bodies if type(b.shape)==Sphere]
numparts = len(sphereIDs);

fluidCoupling.setNumParticles(numparts)
fluidCoupling.setIdList(sphereIDs)
fluidCoupling.isGaussianInterp = False
```

The type of force/velocity interpolation mode has to be set. For Gaussian envelope interpolation, the *isGaussianInterp* flag has to be set, also the solver *pimpleFoamYade* must be used. The engine is added to the O.engines after the timestepper

```
O.engines = [
ForceResetter(),
...,
GlobalStiffnessTimeStepper,
fluidCoupling ...
newton ]
```

Substepping/data exchange interval is set automatically based on the ratio of timesteps as foamDt/yadeDt (see *exchangeDeltaT* for details).

In OpenFOAM

There are two solvers available in this git repository. The solver *icoFoamYade* is based on the point force coupling method and the solver *pimpleFoamYade* is based on the volume averaged coupling. They are based on the existing icoFoam and pimpleFoam solvers respectively. Any OpenFOAM supported mesh can be used, for more details on the mesh options and meshing see here. In the present example, the mesh is generated using *blockMesh* utility of OpenFOAM. The case is set up in the usual OpenFOAM way with the directories *0*, *system* and *constant*

```
0/
  U                       ## velocity boundary conditions
  p                       ## pressure boundary conditions
  uSource                 ## source term bcs (usually set as calculated).

system/
  controlDict             ## simulation settings : start time, end time, delta T,␣
↪solution write control etc.
  blockMeshDict           ## mesh setup for using blockMesh utility : define␣
↪coordinates of geometry and surfaces. (used for simple geometries -> cartesian␣
↪mesh.)
```

(continues on next page)

(continued from previous page)

```
  decomposeParDict          ## dictionary for setting domain decomposition, (in the␣
↪present example scotch is used)
  fvSchemes                 ## selection of finite volume schemes for calculations of␣
↪divergence, gradients and interpolations.
  fvSolution                ## linear solver selection, setting of relaxation factors␣
↪and tolerance criterion,

constant/
  polymesh/                 ## mesh information, generated by blockMesh or other mesh␣
↪utils.
  transportProperties       ## set the fluid and particle properties. (just density␣
↪of the particle)
```

Note: Always set the timestep less than the particle relaxation time scale, this is not claculated automatically yet! Turbulence modelling based on the RANS equations have not been implemented yet. However it is possible to use the present formulations for fully resolved turbulent flow simulations via DNS. Dynamic/moving mesh problems are not supported yet. (Let me know if you're interested in implementing any new features.)

To prepare a simulation, follow these steps:

```
blockMesh         ## generate the mesh
decomposePar      ## decompose the mesh
```

Any type of mesh that is supported by OpenFOAM can be used. Dynamic mesh is currently not supported.

Execution

The simulation is executed via the following command:

```
mpirun -n 4 /path/to/yade/install/bin/yade-exec scriptMPI.py
```

The video below shows the steps involved in compiling and executing the coupled CFD-DEM simulation

4.2.4 Post-Processing

Paraview can be used to visulaize both the Yade solution (use VTKRecorder) and OpenFOAM solution. To visulaize the fluid solution, create an empty file as *name.foam* , open this file in Paraview and in the *properties* section below the pipeline, change "Reconstructed case" to "Decomposed case" , or you can use the reconstructed case itself but after running the *reconstructPar* utility, but this is time consuming.

4.2.5 Using blockMeshDict

The *blockMeshDict* file (*system/blockMeshDict*) can be loaded as facets (*utils.facet*) using the py/ymport.py module's *ymport.blockMeshDict* function:

```
from yade import ymport

facets = ymport.blockMeshDict("system/blockMeshDict")

O.bodies.append(facets)
```

The version of the *blockMeshDict* must be *2.0*, see: py/tests/ymport-files/blockMeshDict.

Only the "boundary" section will be loaded, that is faces `f` consists of vertices `v` in a way that one face is defined by four vertices:

$$f_i = (v_{i0}, v_{i1}, v_{i2}, v_{i3}), \tag{4.17}$$

where vertex v is a point in a three dimensional space:

$$v_{ij} = (x_{ij}, y_{ij}, z_{ij}). \tag{4.18}$$

Two new facets f^* are generated from every face f:

$$f^*_{0i} = (v_{i0}, v_{i1}, v_{i2}), \tag{4.19}$$

$$f^*_{1i} = (v_{i2}, v_{i3}, v_{i0}). \tag{4.20}$$

There are three types of faces: *patch*, *wall* and *empty*. All types are loaded by default, the *patch* and *empty* types can be discarded using the *patchasWall* and *emptyasWall* arguments of *ymport.blockMeshDict*.

4.2.6 Using polyMesh

The *polyMesh* directory (*constant/polyMesh*) can be loaded as facets (*utils.facet*) using the py/ymport.py module's *ymport.polyMesh* function:

```
from yade import ymport

facets = ymport.polyMesh("constant/polyMesh")

O.bodies.append(facets)
```

The function scans the directory and loads the *points*, *faces* and *boundary* files. The files must be *FoamFiles'* with the correct header (version is *2.0*, type is *ascii*, see: py/tests/ymport-files/polyMesh/points). It parses the files and builds the boundary mesh:

The boundary mesh consists of faces f consists of vertices v in a way that one face is defined by four vertices:

$$f_i = (v_{i0}, v_{i1}, v_{i2}, v_{i3}), \tag{4.21}$$

where vertex v is a point in a three dimensional space:

$$v_{ij} = (x_{ij}, y_{ij}, z_{ij}). \tag{4.22}$$

Two new facets f^* are generated from every face f:

$$f^*_{0i} = (v_{i0}, v_{i1}, v_{i2}), \tag{4.23}$$

$$f^*_{1i} = (v_{i2}, v_{i3}, v_{i0}). \tag{4.24}$$

There are three types of faces: *patch*, *wall* and *empty*. All types are loaded by default, the *patch* and *empty* types can be discarded using the *patchAsWall* and *emptyAsWall* arguments of *ymport.polyMesh*.

Note: The *polyMesh* is typically more refined than *blockMeshDict*.

4.3 FEM-DEM hierarchical multiscale modeling with Yade and Escript

Authors: Ning Guo and Jidong Zhao

Institution: Hong Kong University of Science and Technology

Escript download page: https://launchpad.net/escript-finley

mpi4py download page (optional, require MPI): https://bitbucket.org/mpi4py/mpi4py

Tested platforms: Desktop with Ubuntu 10.04, 32 bit; Server with Ubuntu 12.04, 14.04, 64 bit; Cluster with Centos 6.2, 6.5, 64 bit;

4.3.1 Introduction

The code is built upon two open source packages: Yade for DEM modules and Escript for FEM modules. It implements the hierarchical multiscale model (FEMxDEM) for simulating the boundary value problem (BVP) of granular media. FEM is used to discretize the problem domain. Each Gauss point of the FEM mesh is embedded a representative volume element (RVE) packing simulated by DEM which returns local material constitutive responses to FEM. Typically, hundreds to thousands of RVEs are involved in a medium-sized problem which is critically time consuming. Hence parallelization is achieved in the code through either multiprocessing on a supercomputer or mpi4py on a HPC cluster (require MPICH or Open MPI). The MPI implementation in the code is quite experimental. The "mpipool.py" is contributed by Lisandro Dalcin, the author of mpi4py package. Please refer to the examples for the usage of the code.

4.3.2 Finite element formulation

Note

This and the following section are a short excerpt from [Guo2014] to provide some theoretical background. Yade users of FEM-DEM coupling are welcome to improve the following two sections.

In this coupled FEM/DEM framework on hierarchical multiscale modelling of granular media, the geometric domain Ω of a given BVP is first discretised into a suitable FEM mesh. After the finite element discretisation, one ends up with the following equation system to be solved,

$$\mathbf{K}\mathbf{u} = \mathbf{f}, \tag{4.25}$$

where $\mathbf{K}$ is the stiffness matrix, $\mathbf{u}$ is the unknown displacement vector at the FEM nodes and $\mathbf{f}$ is the nodal force vector lumped from the applied boundary traction. For a typical linear elastic problem, $\mathbf{K}$ can be formulated from the elastic modulus, and equation (4.25) can be solved directly. Whilst in the case involving nonlinearity such as for granular media where $\mathbf{K}$ depends on state parameters and loading history, Newton–Raphson iterative method needs to be adopted and the stiffness matrix is replaced with the tangent matrix $\mathbf{K}_t$, which is assembled from the tangent operator:

$$\mathbf{K}_t = \int_\Omega \mathbf{B}^T \mathbf{D} \mathbf{B} dV, \tag{4.26}$$

where $\mathbf{B}$ is the deformation matrix (i.e. gradient of the shape function), and $\mathbf{D}$ is the matrix form of the rank four tangent operator tensor $\mathbf{D}$. During each Newton–Raphson iteration, both $\mathbf{K}_t$ and internal stress $\boldsymbol{\sigma}$ are updated, and the scheme tries to minimise the residual force $\mathbf{R}$ to find a converged solution:

$$\mathbf{R} = \int_\Omega \mathbf{B}^T \boldsymbol{\sigma} dV - \mathbf{f}. \tag{4.27}$$

The tangent operator and the stress tensor at each local Gauss integration point are pivotal variables in the aforementioned calculation and need to be evaluated before each iteration and loading step. A continuum-based conventional FEM usually assumes a constitutive relation for the material and derives

the tangent matrix and the stress increment based on this constitutive assumption (e.g. using the elasto-plastic modulus $\mathbf{D}^{ep}$ in equation (4.26) to assemble $\mathbf{K}_t$ and to integrate stress). The coupled FEM/DEM multiscale approach obtains the two quantities from the embedded discrete element assembly at each Gauss point and avoids the needs for phenomenological assumptions.

4.3.3 Multiscale solution procedure

The hierarchical multiscale modelling procedure is schematically summarised in the following steps:

1. Discretise the problem domain by suitable FEM mesh and attach each Gauss point with a DEM assembly prepared with suitable initial state.
2. Apply one global loading step, that is, imposed by FEM boundary condition on $\partial\Omega$.
   a) Determine the current tangent operator for each RVE.
   b) Assemble the global tangent matrix using equation (4.26) and obtain a trial solution of displacement $\mathbf{u}$ by solving Equation (4.25) with FEM.
   c) Interpolate the deformation $\nabla\mathbf{u}$ at each Gauss point of the FEM mesh and run the DEM simulation for the corresponding RVE using $\nabla\mathbf{u}$ as the DEM boundary conditions.
   d) Derive the updated total stress for each RVE and use it to evaluate the residual by equation (4.27) for the FEM domain.
   e) Repeat the aforementioned steps from (a) to (d) until convergence is reached and finish the current loading step.
3. Proceed to the next loading step and repeat Step 2.

In interpolating the deformation $\mathbf{u}$ from the FEM solution for DEM boundary conditions in Step 2(c), we consider both the infinitesimal strain $\boldsymbol{\varepsilon}$ and rotation $\boldsymbol{\omega}$

$$\nabla\mathbf{u} = \underbrace{\frac{1}{2}(\nabla\mathbf{u} + \nabla\mathbf{u}^{\mathsf{T}})}_{\boldsymbol{\varepsilon}} + \underbrace{\frac{1}{2}(\nabla\mathbf{u} - \nabla\mathbf{u}^{\mathsf{T}})}_{\boldsymbol{\omega}} \tag{4.28}$$

The corresponding RVE packing will deform according to this prescribed boundary condition.

It is also instructive to add a few remarks on the evolution of stress from the RVE in Step 2(d). In traditional FEM, the stress is updated based on an incremental manner to tackle the nonlinear material response. If small strain is assumed, the incremental stress–strain relation may potentially cause inaccurate numerical results when large deformation occurs in the material, which calls for an alternative formulation for large deformation. This issue indeed can be naturally circumvented in the current hierarchical framework. In our framework, the DEM assembly at each Gauss point will memorise its past state history (e.g. pressure level, void ratio and fabric structure) and will be solved with the current applied boundary condition (including both stretch and rotation) at each loading and iteration step. Towards the end of each loading step, instead of using an incremental stress update scheme, the total true stress (Cauchy stress) is derived directly over the solved DEM assembly through homogenisation and is then returned to the FEM solver for the global solution. In this way, we do not have to resort to the use of other objective stress measures to deal with large deformation problems. However, we note that a proper strain measurement is still required and the FEM mesh should not be severely distorted, otherwise, remeshing of the FEM domain will be required.

More detailed description of the solution procedure can be found in [Guo2013], [Guo2014], [Guo2014b], [Guo2014c], [Guo2015].

4.3.4 Work on the YADE side

The version of YADE should be at least rev3682 in which Bruno added the stringToScene function. Before installation, I added some functions to the source code (in "yade" subfolder). But only one function ("Shop::getStressAndTangent" in "./pkg/dem/Shop.cpp") is necessary for the FEMxDEM coupling, which returns the stress tensor and the tangent operator of a discrete packing. The former is homogenized using the Love's formula and the latter is homogenized as the elastic modulus. After installation and we get the executable file: yade-versionNo. We then generate a .py file linked to the executable

file by "ln yade-versionNo yadeimport.py". This .py file will serve as a wrapped library of YADE. Later on, we will import all YADE functions into the python script through "from yadeimport import *" (see simDEM.py file).

Open a python terminal. Make sure you can run

```
import sys
sys.path.append('where you put yadeimport.py')
from yadeimport import *
Omega().load('your initial RVE packing, e.g. 0.yade.gz')
```

If you are successful, you should also be able to run

```
from simDEM import *
```

4.3.5 Work on the Escript side

No particular requirement. But make sure the modules are callable in python, which means the main folder of Escript should be in your PYTHONPATH and LD_LIBRARY_PATH. The modules are wrapped as a class in msFEM*.py.

Open a python terminal. Make sure you can run:

```
from esys.escript import *
from esys.escript.linearPDEs import LinearPDE
from esys.finley import Rectangle
```

(Note: Escript is used for the current implementation. It can be replaced by any other FEM package provided with python bindings, e.g. FEniCS (http://fenicsproject.org). But the interface files "msFEM*.py" need to be modified.)

4.3.6 Example tests

After Steps 1 & 2, one should be able to run all the scripts for the multiscale analysis. The initial RVE packing (default name "0.yade.gz") should be provided by the user (e.g. using YADE to prepare a consolidated packing), which will be loaded by simDEM.py when the problem is initialized. The sample is initially uniform as long as the same RVE packing is assigned to all the Gauss points in the problem domain. It is also possible for the user to specify different RVEs at different Gauss points to generate an inherently inhomogeneous sample.

While simDEM.py is always required, only one msFEM*.py is needed for a single test. For example, in a 2D (3D) dry test, msFEM2D.py (msFEM3D.py) is needed; similarly for a coupled hydro-mechanical problem (2D only, saturated), msFEMup.py is used which incorporates the u-p formulation. Multiprocessing is used by default. To try MPI parallelization, please set useMPI=True when constructing the problem in the main script. Example tests given in the "example" subfolder are listed below. Note: The initial RVE packing (named 0.yade.gz by default) needs to be generated, e.g. using prepareRVE.py in "example" subfolder for a 2D packing (similarly for 3D).

1. **2D drained biaxial compression test on dry dense sand** (biaxialSmooth.py) *Note*: Test description and result were presented in [Guo2014] and [Guo2014c].
2. **2D passive failure under translational mode of dry sand retained by a rigid and frictionless wall** (retainingSmooth.py) *Note:* Rolling resistance model (CohFrictMat) is used in the RVE packing. Test description and result were presented in [Guo2015].
3. **2D half domain footing settlement problem with mesh generated by Gmsh** (footing.py, footing.msh) *Note:* Rolling resistance model (CohFrictMat) is used in the RVE packing. Six-node triangle element is generated by Gmsh with three Gauss points each. Test description and result were presented in [Guo2015].
4. **3D drained conventional triaxial compression test on dry dense sand using MPI parallelism** (triaxialRough.py) *Note 1:* The simulation is very time consuming. It costs ~4.5 days on

one node using multiprocessing (16 processes, 2.0 GHz CPU). When useMPI is switched to True (as in the example script) and four nodes are used (80 processes, 2.2 GHz CPU), the simulation costs less than 24 hours. The speedup is about 4.4 in our test. *Note 2:* When MPI is used, mpi4py is required to be installed. The MPI implementation can be either MPICH or Open MPI. The file "mpipool.py" should also be placed in the main folder. Our test is based on openmpi-1.6.5. This is an on-going work. Test description and result will be presented later.

5. **2D globally undrained biaxial compression test on saturated dense sand with changing permeability using MPI parallelism** (undrained.py) *Note:* This is an on-going work. Test description and result will be presented later.

4.3.7 Disclaim

This work extensively utilizes and relies on some third-party packages as mentioned above. Their contributions are acknowledged. Feel free to use and redistribute the code. But there is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

4.4 Simulating Acoustic Emissions in Yade

Suggested citations:

Caulk, R. (2018), Stochastic Augmentation of the Discrete Element Method for Investigation of Tensile Rupture in Heterogeneous Rock. *Yade Technical Archive*. DOI 10.5281/zenodo.1202039. download full text

Caulk, Robert A. (2020), Modeling acoustic emissions in heterogeneous rocks during tensile fracture with the Discrete Element Method. Open Geomechanics, Volume 2, article no. 2, 19 p. doi : 10.5802/ogeo.5. full text

4.4.1 Summary

This document briefly describes the simulation of acoustic emissions (AE) in Yade. Yade's clustered strain energy based AE model follows the methods introduced by [Hazzard2000] and [Hazzard2013]. A validation of Yade's method and a look at the effect of rock heterogeneity on AE during tensile rock failure is discussed in detail in [Caulk2018] and [Caulk2020].

4.4.2 Model description

Numerical AE events are simulated by assuming each broken bond (or cluster of broken bonds) represents an event location. Additionally, the associated system strain energy change represents the event magnitude. Once a bond breaks, the strain energies (E_i) are summed for all intact bonds within a predefined spatial radius (λ):

$$E_i = \frac{1}{2}\left(\frac{F_n^2}{k_n} + \frac{F_s^2}{k_s}\right)$$

$$E_o = \sum_i^N E_i$$

where F_n, F_s and k_n, k_s are the normal and shear force (N) and stiffness (N/m) components of the interaction prior to failure, respectively. Yade's implementation uses the maximum change of strain energy surrounding each broken bond to estimate the moment magnitude of the AE. As soon as the bond breaks, the total strain energy ($E_o = \sum_i^N E_i$) is computed for the radius (set by the user as no. of avg particle diameters, λ. E_o is used as the reference strain energy to compute $\Delta E = E - E_o$ during subsequent time steps. Finally, $\max(\Delta E)$ is used in the empirical equation derived by [Scholz2003]:

$$M_e = \frac{2}{3}\log \Delta E - 3.2$$

Events are clustered if they occur within spatial and temporal windows of other events, similar to the approach presented by [Hazzard2000] and [Hazzard2013]. The spatial window is simply the user defined λ and the temporal window T_{max} is computed as:

$$T_{max} = int\left(\frac{D_{avg}\lambda}{max(v_{p1}, v_{p2})\Delta t}\right)$$

where D_{avg} is the average diameter of the particles comprising the failed event (m), v_{p1} and v_{p2} are the P-Wave velocities (m/s) of the particle densities, and Δt is the time step of the simulation (seconds/time step). As shown in *fig-cluster*, the final location of a clustered event is simply the average of the clustered event centroids. Here the updated reference strain energy is computed by adding the strain energy of the unique interactions surrounding the new broken bond to the original reference strain energy (E_o):

- Original bond breaks, sum strain energy of broken bonds (N_{orig}) within spatial window $E_{orig,o} = \sum_{i=1}^{N_{orig}} E_i$
- New broken bond detected within spatial and temporal window of original bond break
- Update reference strain E_o by adding unique bonds (N_{new}) within new broken bond spatial window $E_{new,o} = E_{orig,o} + \sum_{i=1}^{N_{new}} E_i$

This method maintains a physical reference strain energy for the calculation of $\Delta E = E - E_{new,o}$ and depends strongly on the spatial window size. Ultimately, the clustering increases the number of larger events, which yields more comparable b-values to typical Guttenberg Richter curves [Hazzard2013].

For a detailed look at the underlying algorithm, please refer to the source code.

4.4.3 Activating the algorithm within Yade

The simulation of AE is available as part of Yade's Jointed Cohesive Frictional particle model *(JCFpm)* . As such, your simulation needs to make use of *JCFpmMat* , *JCFpmPhys* , and *Law2_ScGeom_-JCFpmPhys*

Your material assignment and engines list might look something like this:

```
      JCFmat = O.materials.append(JCFpmMat(young=young, cohesion=cohesion,
              density=density, frictionAngle=radians(finalFricDegree),
              tensileStrength=sigmaT, poisson=poisson, label='JCFmat',
              jointNormalStiffness=2.5e6,jointShearStiffness=1e6,jointCohesion=1e6))

      O.engines=[
                      ForceResetter(),
                      InsertionSortCollider([Bo1_Box_Aabb(),Bo1_Sphere_Aabb
                              ,Bo1_Facet_Aabb()]),
                      InteractionLoop(
                              [Ig2_Sphere_Sphere_ScGeom(),␣
↪Ig2_Facet_Sphere_ScGeom()],
                              [Ip2_FrictMat_FrictMat_FrictPhys(),
                                      Ip2_JCFpmMat_JCFpmMat_JCFpmPhys( \
                                                                  ␣
↪xSectionWeibullScaleParameter=xSectionScale,
                                                                  ␣
↪xSectionWeibullShapeParameter=xSectionShape,
                                                      weibullCutOffMin=weibullCutOffMin,
                                                      weibullCutOffMax=weibullCutOffMax)],
                              [Law2_ScGeom_JCFpmPhys_JointedCohesiveFrictionalPM(\
                                      recordCracks=True, recordMoments=True,
                                      Key=identifier,label='interactionLaw'),
                              Law2_ScGeom_FrictPhys_CundallStrack()]
),
```

(continues on next page)

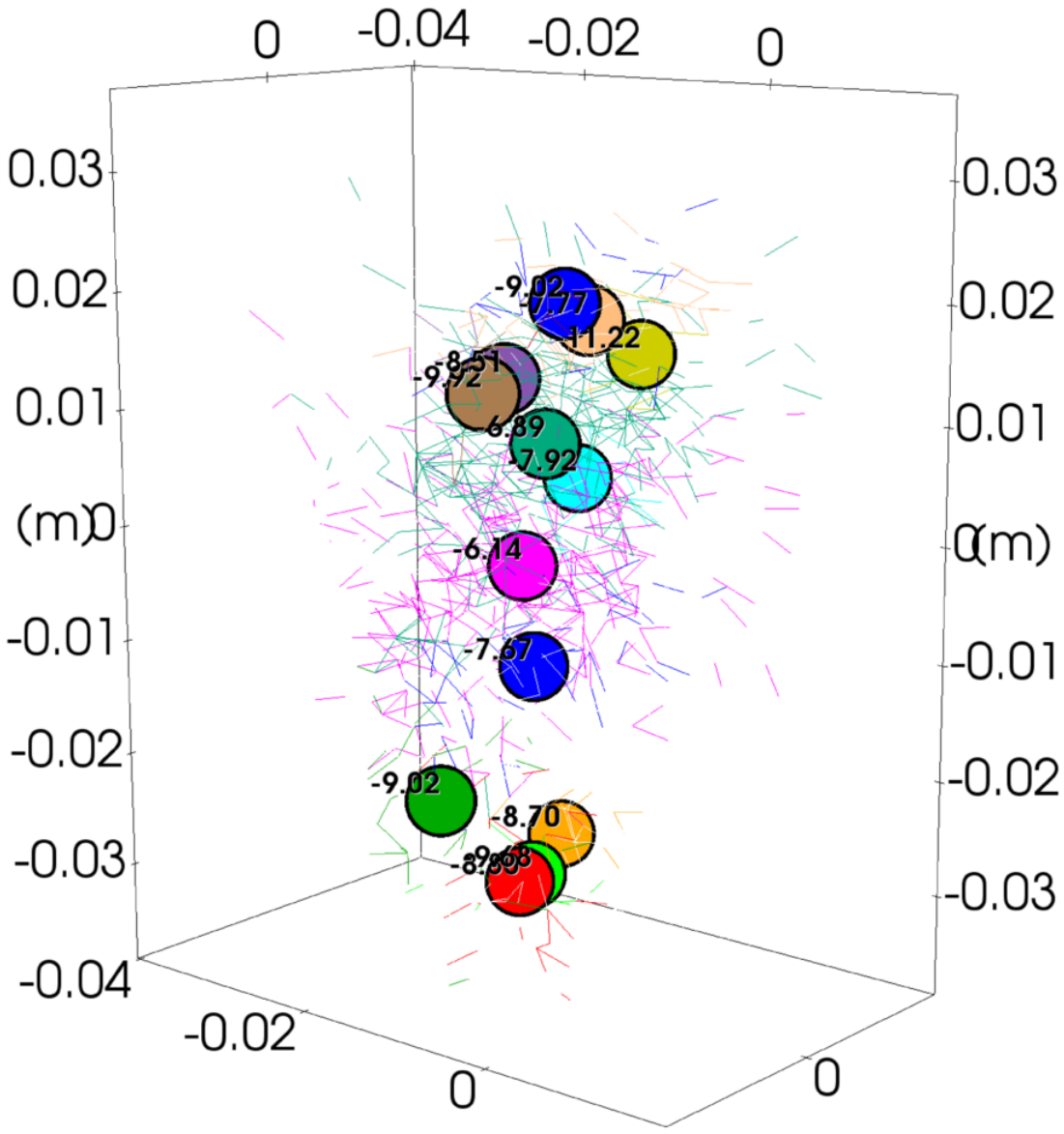


Fig. 1: Example of clustered broken bonds (colored lines) and the final AE events (colored circles) with their event magnitudes.

(continued from previous page)

```
        GlobalStiffnessTimeStepper(),
        VTKRecorder(recorders=['jcfpm','cracks','facets','moments'] \
                            ,Key=identifier,label='vtk'),
                    NewtonIntegrator(damping=0.4)
    ]
```

Most of this simply enables JCFpm as usual, the AE relevant commands are:

```
Law2_ScGeom_JCFpmPhys_JointedCohesiveFrictionalPM(...  recordMoments=True ...)
VTKRecorder(... recorders=[... 'moments' ...])
```

There are some other commands necessary for proper activation and use of the acoustic emissions algorithm:

clusterMoments tells Yade to cluster new broken interactions within the user set spatial radius as described above in the model description. This value is set to True by default.

momentRadiusFactor is λ from the above model description. The momentRadiusFactor changes the number of particle radii beyond the initial interaction that Yade computes the strain energy change. Additionally, Yade uses λ to seek additional broken bonds for clustering. This value is set to 5 by default ([Hazzard2013] concluded that this value yields accurate strain energy change approximations for the total strain energy change of the system entire system).

neverErase allows old interactions to be stored in memory despite no longer affecting the simulation. This value must be set to True for stable operation of Yade's AE cluster model.

4.4.4 Visualizing and post processing acoustic emissions

AE are visualized and post processed in a similar manner to JCFpm cracks. As long as *recordMoments=True* and *recorder=['moments']* , the simulation will produce timestamped .vtu files for easy Paraview post processing. Within Paraview, the *AE can be filtered according to magnitude, number of constitiuent interactions, and event time. fig-aeexample* shows AE collected during a three point bending test and filtered according to magnitude and time

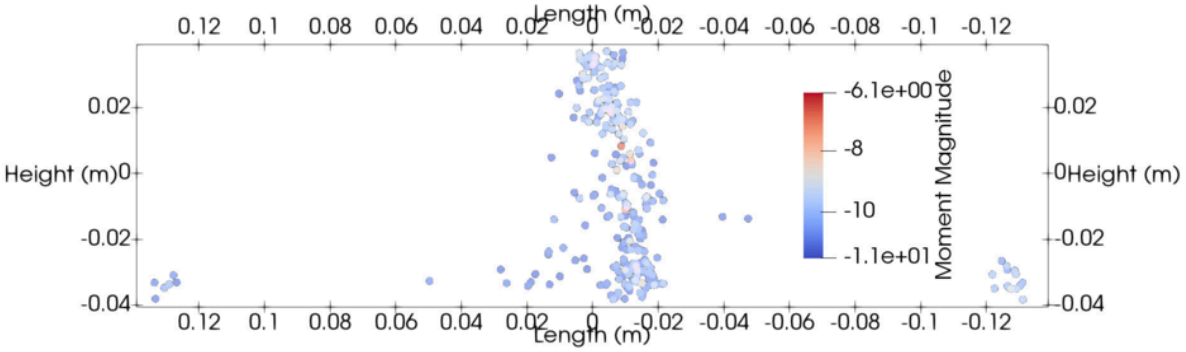


Fig. 2: Example of AE simulated during three point bending test and filtered by magnitude and time.

4.4.5 Consideration of rock heterogeneity

[Caulk2018] and [Caulk2020] hypothesize that heterogeneous rock behavior depends on the distribution of interacting grain edge lengths. In support of the hypothesis, [Caulk2018] and [Caulk2020] show how rock heterogeneity can be modeled using cathodoluminescent grain imagery. A Weibull distribution is constructed based on the so called grain edge interaction length distribution. In Yade's *JCFpm* , the Weibull distribution is used to modify the interaction strengths of contacting particles by correcting the interaction area A_{int}:

$$A_{int} = \pi(\alpha_w \times \min(R_a, R_b))^2$$

where α_w is the Weibull correction factor, which is distributed as shown in *fig-weibullDist*. The corresponding tensile strength distributions for various Weibull shape parameters are shown in *fig-strengthDist*. Note: a Weibull shape factor of ∞ is equivalent to the unaugmented JCFpm model.

In Yade, the application of rock heterogeneity is as simple as passing a Weibull shape parameter to *JCFpmPhys* :

```
Ip2_JCFpmMat_JCFpmMat_JCFpmPhys(
        xSectionWeibullScaleParameter=xSectionScale,
        xSectionWeibullShapeParameter=xSectionShape,
        weibullCutOffMin=weibullCutOffMin,
        weibullCutOffMax=weibullCutOffMax)
```

where the *xSectionWeibullShapeParameter* is the desired Weibull shape parameter. The scale parameter can be assigned in similar fashion. If you want to control the minimum allowable correction factor, you can feed it *weibullCutoffMin* . The maximum correction factor can be controlled in similar fashion.

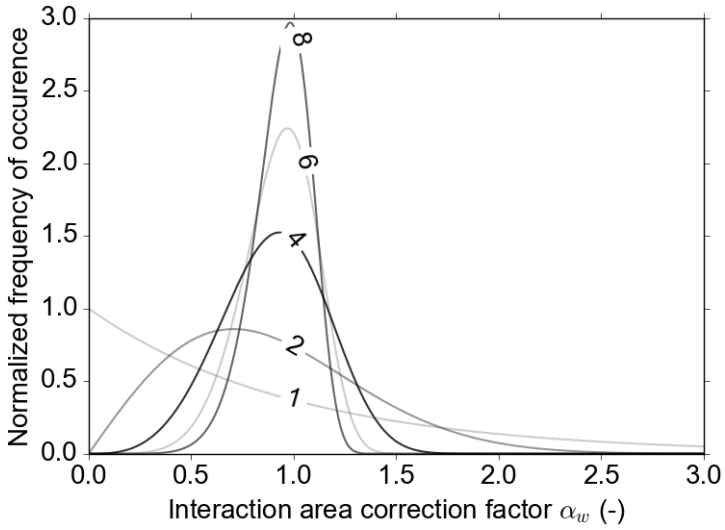


Fig. 3: Weibull distributions for varying shape parameters used to generate α_w.

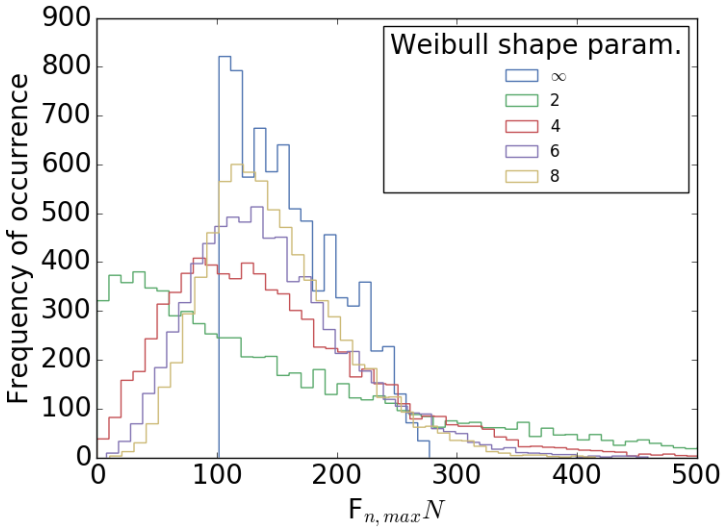


Fig. 4: Maximum DEM particle bond tensile strength distributions for varying Weibull shape parameters.

4.5 Using YADE 1D vertical VANS fluid resolution

The goal of the present note is to detail how the DEM-fluid coupling can be used in practice in YADE. It is complementary with the three notes [Maurin2018_VANSbasis], [Maurin2018_VANSfluidResol] and [Maurin2018_VANSvalidations] detailing respectively the theoretical basis of the fluid momentum balance equation, the numerical resolution, and the validation of the code.

All the coupling and the fluid resolution relies only on the engine HydroForceEngine, which use is detailed here. Examples scripts using HydroForceEngine for different purposes can be found in YADE source code in the folder trunk/examples/HydroForceEngine/. In order to get familiar with this engine, it is recommended to read the present note and test/modify the examples scripts.

4.5.1 DEM-fluid coupling and fluid resolution in YADE

In YADE, the fluid coupling with the DEM is done through the engine called HydroForceEngine, which is coded in the source in the files trunk/pkg/common/HydroForceEngine.cpp and hpp. HydroForceEngine has three main functions:

- It applies drag and buoyancy to each particle from a 1D vertical fluid velocity profile (HydroForceEngine::action)
- It can evaluates the average drag force, particle velocity and solid volume fraction profiles (HydroForceEngine::averageProfile)
- It can solves the fluid velocity equation detailed in the first section, from given average drag force, particle velocity and solid volume fraction profiles (HydroForceEngine::fluidResolution)

We clearly see the link between the three functions. The idea is to evaluate the average profiles from the DEM, put it as input to the fluid resolution, and apply the fluid forces corresponding to the obtained fluid velocity profile to the particles. In the following, the three points will be detailed separately with precision and imaging with the example scripts available in yade source code at trunk/examples/HydroForceEngine/.

4.5.2 Application of drag and buoyancy forces (HydroForceEngine::action)

By default, when adding HydroForceEngine to the list of engine, it applies drag and buoyancy to all the particles which IDs have been passed in argument to HydroForceEngine through the ids variable. This is done for example, in the example script trunk/examples/HydroForceEngine/, in the engine lists:

```
O.engines = [
ForceResetter(),
...
HydroForceEngine(densFluid = densFluidPY,...,ids = idApplyForce),
...
NewtonIntegrator(gravity=gravityVector, label='newtonIntegr')
]
```

where `idApplyForce` corresponds to a list of particle ID to which the hydrodynamic forces should be applied. The expression of the buoyancy and drag force applied to the particles contained in the id list is detailed below.

In case where the fluid is at rest (HydroForceEngine.steadyFlow = False), HydroForceEngine applies buoyancy on a particle p from the fluid density and the acceleration of gravity g as:

$$\mathbf{f}_b^p = -\rho^f V^p \mathbf{g}.$$

Meanwhile, if the fluid flow is steady and turbulent, the buoyancy which is related to the fluid pressure gradient does not have a term in the streamwise direction (see discussion p. 5 of [Maurin2018]). Puting the option HydroForceEngine.steadyFlow to True turns the expression of the buoyancy into:

$$\mathbf{f}_b^p = -\rho^f V^p (\mathbf{g}.\mathbf{e}_x)\mathbf{e}_x.$$

Also, HydroForceEngine applies a drag force to each particles contained in the ids list. This drag force depends on the velocity of the particles and on the fluid velocity, which is defined by a 1D fluid velocity profile, HydroForceEngine.vxFluid. This fluid velocity profile can be evaluated from the fluid model, but can also be imposed by the user and stay constant. From this 1D vertical fluid velocity profile, the drag force applied to particle p reads:

$$\mathbf{f}_D^p = \frac{1}{2} C_d A \rho^f \|u_p^f \mathbf{e}_x - \mathbf{v}^p\| \left(u_p^f \mathbf{e}_x - \mathbf{v}^p\right),$$

where $\mathbf{u}_p^f$ is the fluid velocity at the center of particle p, $\mathbf{v}^p$ is the particle velocity, ρ^f is the fluid density, $A = \pi d^2/4$ is the area of the sphere submitted to the flow, and C_d is the drag coefficient accounts for the effects of particle Reynolds number [Dallavalle1948] and of increased drag due to the presence of other particles (hindrance, [Richardson1954]:

$$C_d = \left(0.44 + \frac{24}{Re_p}\right)(1-\varphi_p)^{-\gamma} = \left(0.44 + 24\frac{\nu^f}{\|u_p^f \mathbf{e}_x - \mathbf{v}^p\| d}\right)(1-\varphi_p)^{-\gamma}$$

with φ_p the solid volume fraction at the center of the particle evaluated from HydroForceEngine.phiPart, and γ the Richardson-Zaki exponent, which can be set through the parameter HydroForceEngine.expoRZ (3.1 by default).

HydroForceEngine can also apply a lift force, but this is not done by default (HydroForceEngine.lift = False), and this is not recommended by the author considering the uncertainty on the actual formulation (see discussion p. 6 of [Maurin2015] and [Schmeeckle2007]).

As the fluid velocity profile (HydroForceEngine.vxFluid) and solid volume fraction profile (HydroForceEngine.phiPart) can be imposed by the user, the application of drag and buoyancy to the particles through HydroForceEngine can be done without using the function averageProfile and the fluid resolution. Examples of such use can be found in the source code: trunk/examples/HydroForceEngine/oneWayCouplingfootnote{In this case, we talk about a one-way coupling as the fluid influence the particles but is not influenced back}.

4.5.3 Solid phase averaging (HydroForceEngine::averageProfile)

In order to solve the fluid equation, we have seen that it is necessary to compute from the DEM the solid volume fraction, the solid velocity, and the averaged drag profiles. The function HydroForceEngine.averageProfile() has been set up in order to do so. It is designed to evaluate the average profiles over a regular grid, at the position between two mesh nodes. In order to match the fluid velocity profile numerotation, the averaged vector are of size $ndimz + 1$ even though the quantities at the top and bottom boundaries are not evaluated and set to zero by defaultfootnote{It is not necessary to evaluate the solid DEM quantities at the boundaries are they are not considered in the fluid resolution, see subsection boundaries of [Maurin2018_VANSfluidResol]}. textcolor{red}{You should do that}

The solid volume fraction profile is evaluated by considering the volume of particles contained in the layer considered. The layer is defined by the mesh step along the wall-normal direction, but extend over the whole length and width of the sample. We perform such an averaging only discretized over the wall-normal direction in order to match the fluid resolution. Meanwhile, this is also physical as, at steady state the problem is unidirectional on average, so that the only variation we should observe in the measured averaged quantities should be along the vertical direction, z. Therefore, the solid volume fraction is evaluated by considering the volume of particles which is contained inside the layer considered $i + 1/2$:

$$\varphi_{i+1/2} = \sum_{p \in [i\,dz;(i+1)\,dz]} V^p_{i+1/2};$$

where the sum is over the particles p which have at least a part of their volume inside the layer $i + 1/2$, i.e. in between an elevation of $i * dz$ and $(i+1) * dz$, and $V^p_{i+1/2}$ is the volume of the particles considered which is contained inside the layer considered. The latter correspond to the integral between two points of a slice of sphere and can be evaluated analytically in cylindrical coordinate. Following this formulation and the formalism of [Jackson2000] with a weighting step function, any particle-associated quantity K

can be averaged with the following formulation:

$$\langle K\rangle^p\big|_{i+1/2} = \frac{\sum_{p\in[idz;(i+1)dz]} V^p_{i+1/2} K^p}{\sum_{p\in[idz;(i+1)dz]} V^p_{i+1/2}},$$

Where K^p is the quantity associated with particle p, e.g. the particle streamwise velocity. In this case, we can write:

$$\langle v_x\rangle^p|_{i+1/2} = \frac{\sum_{p\in[idz;(i+1)dz]} V^p_{i+1/2} v^p_x}{\sum_{p\in[idz;(i+1)dz]} V^p_{i+1/2}},$$

where v^p_x is the velocity of particle p. Regarding the evaluation of the average streamwise drag force transmitted by the fluid to the particles, it can be written similarly as:

$$\langle f_{D,x}\rangle^p|_{i+1/2} = \frac{\sum_{p\in[idz;(i+1)dz]} V^p_{i+1/2} f^p_{D,x}}{\sum_{p\in[idz;(i+1)dz]} V^p_{i+1/2}},$$

where $f^p_{D,x}$ is the drag force on particle p.

As will be detailed in the next part, these averaged profile can be used for the fluid resolution, but they can also be used for analysis as done for example for bedload transport in [Maurin2015b] [Maurin2018].

4.5.4 Fluid resolution\HydroForceEngine::fluidResolution

In order to use the fluid resolution inside the fluid-DEM coupling framework, it is necessary to call the function HydroForceEngine.averageProfile() in order to evaluate the averaged solid volume fraction profile, streamwise velocity and streamwise drag force. The latter is necessary in order to evaluate the terms β taken into account in the fluid equation (see [Maurin2018_VANSfluidResol] for details). β is defined as:

$$n\left\langle f^f_x\right\rangle^p\Big|_{i+1/2} = \beta_{i+1/2}\left(\left\langle u_x\right\rangle^f\Big|_{i+1/2} - \left\langle v_x\right\rangle^p\Big|_{i+1/2}\right)$$

so that it can be evaluated directly from the averaged drag, particle velocity and the fluid velocity at the last iteration (explicited the term β in the fluid resolution):

$$\beta^n_{i+1/2} = \frac{n\left\langle f^f_x\right\rangle^p\Big|^{n-1}_{i+1/2}}{\left\langle u_x\right\rangle^f\Big|^{n-1}_{i+1/2} - \left\langle v_x\right\rangle^p\Big|^{n-1}_{i+1/2}}$$

where the solid variables have been denoted with a superscript $n-1$ as they are known and not re-evaluated at each time stepfootnote{In a way β^n should probably be better written as β^{n-1}}. This terms is called taufsi and is directly evaluated inside the code.

All the quantities needed in order to solve the fluid resolution - highlighted in [Maurin2018_VANSfluidResol] and recalled in figure *fig-scheme* - are now explicited. They can be directly evaluated in YADE with the function HydroForceEngine.averageProfile(). From there, the fluid resolution can be performed over a given time t_{resol} with a given time step Δt by calling directly the function HydroForceEngine.fluidResolution (t_{resol},Δt). This will perform the fluid resolution described in [Maurin2018_VANSfluidResol], $N = t_{resol}/\Delta t$ times, with a time step Δt, considering the vertical profiles of β, $\langle v_x\rangle$ and φ as constant in time. Therefore, one should not only be carefull about the time step, but also about the period of coupling, which should not be too large in order to avoid unphysical behavior in the DEM due to a drastic change of velocity profile not compensated by an increased transmitted drag force.

In the example script in YADE source code, trunk/examples/HydroForceEngine/twoWayCoupling/sedimentTransportExa
1DRANSCoupling.py, the DEM and fluid resolution are coupled with a period of $\texttt{fluidResolPeriod} = 10^{-2}$s by default, and with a fluid time step of $\texttt{dtFluid} = 10^{-5}$s. This means that the DEM is let evolved for 10^{-2}s, and frozen during the fluid resolution which is made over $\texttt{fluidResolPeriod/dtFluid} = 10^3$ step with $\Delta t = 10^{-5}$. Then, the DEM is let evolved again but with a new fluid velocity profile for 10^{-2}s,

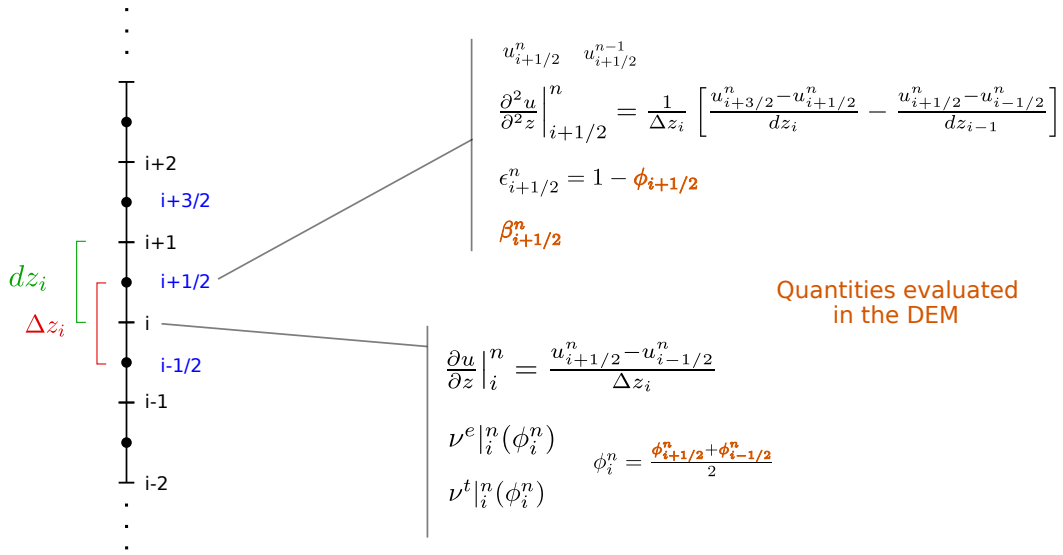


Fig. 5: Schematical picture of the numerical fluid resolution and variables definition with a regular mesh. All the definitions still holds for a mesh with variable spatial step.

and frozen...etc. This period between two fluid resolution should be tested and taken not too long (see appendix of [Maurin2015b]).

Meanwhile, the fluid resolution can be used in itself, without DEM coupling, in particular to verify the fluid resolution in known cases. This is done in the example folder of YADE source code, trunk/examples/HydroForceEngine/fluidValidation/, where the cases of a poiseuille flow and a log layer have been considered and validated.

4.6 Potential Particles and Potential Blocks

The origins of scientific development regarding the algorithms described in this section are traced back to: [Boon2012] (*Potential Blocks* code), [Boon2013b] (*Potential Particles* code) and [Boon2015] (*Block Generation* code).

4.6.1 Introduction

This section discusses two codes to simulate (i) non-spherical particles using the concept of the Potential Particles [Houlsby2009], with the solution procedures in [Boon2013] for 3-D and (ii) polyhedral blocks using planar linear inequalities, based on linear programming concepts [Boon2012]. These codes define two shape classes in YADE, namely *PotentialParticle* and *PotentialBlock*. Besides some similarities in syntax, they have distinct differences, concerning morphological characteristics of the particles and the methods used to facilitate contact detection.

The *Potential Particles* code (abbreviated herein as *PP*) is detailed in [Boon2013], where non-spherical particles are assembled as a combination of 2nd degree polynomial functions and a fraction of a sphere, while their edges are rounded with a user-defined radius of curvature.

The *Potential Blocks* code (abbreviated herein as *PB*) is used to simulate polyhedral particles with flat surfaces, based on the work of [Boon2012], where a smooth, inner potential particle is used to calculate the contact normal vector. This code is compatible with the *Block Generation* algorithm defined in [Boon2015], in which Potential Blocks can be generated by intersections of original, intact blocks with discontinuity planes.

These two codes are independent, in the sense that either one of them can be compiled/used separately, without enabling the other, while they do not interact with each other (i.e. we cannot establish contact between a PP and a PB). Enabling the PB code causes an automatic compilation of the *Block Generation* algorithm.

4.6.2 Potential Particles code (PP)

The concept of *Potential Particles* was introduced and developed by [Houlsby2009]. The problem of contact detection between a pair of potential particles was cast as a constrained optimization problem, where the equations are solved using the Newton-Raphson method in 2-D. In [Boon2013] it was extended to 3-D and more robust solutions were proposed. Many numerical optimization solvers generally cannot cope with discontinuities, ill-conditioned gradients (Jacobians) or curvatures (Hessians), and these obstacles were overcome in [Boon2013], by re-formulating the problem and solving the equations using conic optimization solvers. Previous versions made use of MOSEK (using its academic licence), while currently an in-house code written by [Boon2013] is used to solve the conic optimization problem. A potential particle is defined as in (4.29) [Houlsby2009]:

$$f = (1-k)\left(\sum_{i=1}^{N}\langle a_i x + b_i y + c_i z - d_i\rangle^2 - r^2\right) + k(x^2+y^2+z^2-R^2) \tag{4.29}$$

where (a_i, b_i, c_i) is the normal vector of the ith plane, defined with respect to the particle's local coordinate system and d_i is the distance of the plane to the local origin. $\langle\ \rangle$ are Macaulay brackets, i.e., $\langle x\rangle = x$ for $x > 0$; $\langle x\rangle = 0$ for $x \leq 0$. The planes are assembled such that their normal vectors point outwards. They are summed quadratically and expanded by a distance r, which is also related to the radius of the curvature at the corners. Furthermore, a "shadow" spherical particle is added; R is the radius of the sphere, with $0 < k \leq 1$, denoting the fraction of sphericity of the particle. The geometry of some cuboidal potential particles is displayed in Fig. *fig-pp*, for different values of the parameter k.

The potential function is normalized for computational reasons in the form (4.30) [Houlsby2009]:

$$f = (1-k)\left(\sum_{i=1}^{N}\frac{\langle a_i x + b_i y + c_i z - d_i\rangle^2}{r^2} - 1\right) + k\left(\frac{x^2+y^2+z^2}{R^2} - 1\right) \tag{4.30}$$

This potential function takes values:

- $f = 0$: on the particle surface
- $f < 0$: inside the particle
- $f > 0$: outside the particle

To ensure numerical stability, it is not advised to use values approaching *k=0*. In particular, the extreme value *k=0* cannot be used from a theoretical standpoint, since the *Potential Particles* were formulated for strictly convex shapes (curved faces).

4.6.3 Potential Blocks code (PB)

The *Potential Blocks* code was developed during the D.Phil. thesis of CW Boon [Boon2013b] and discussed in [Boon2012]. It was developed originally for rock engineering applications, to model polygonal and polyhedral blocks with flat surfaces. The blocks are defined with linear inequalities only and unlike the *PotentialParticle* shape class, no spherical term is considered (so, practically k=0). Although k and R are input parameters of the *PotentialBlock* shape class, their existence during computation is null. In particular, R is used within the source code, denoting a characteristic dimension of the blocks, but does not reflect the radius of a "shadow particle", like it does for the *Potential Particles*. This value of R is used in the *Potential Blocks* code to calculate the initial bi-section step size for line search, to obtain a point on the particle, which in turn is used to calculate the overlap distance during contact.

For a convex particle defined by N planes, the space that it occupies can be defined using the following inequalities (4.31):

$$a_i x + b_i y + c_i z \ \leq\ d_i, i = 1:N \tag{4.31}$$

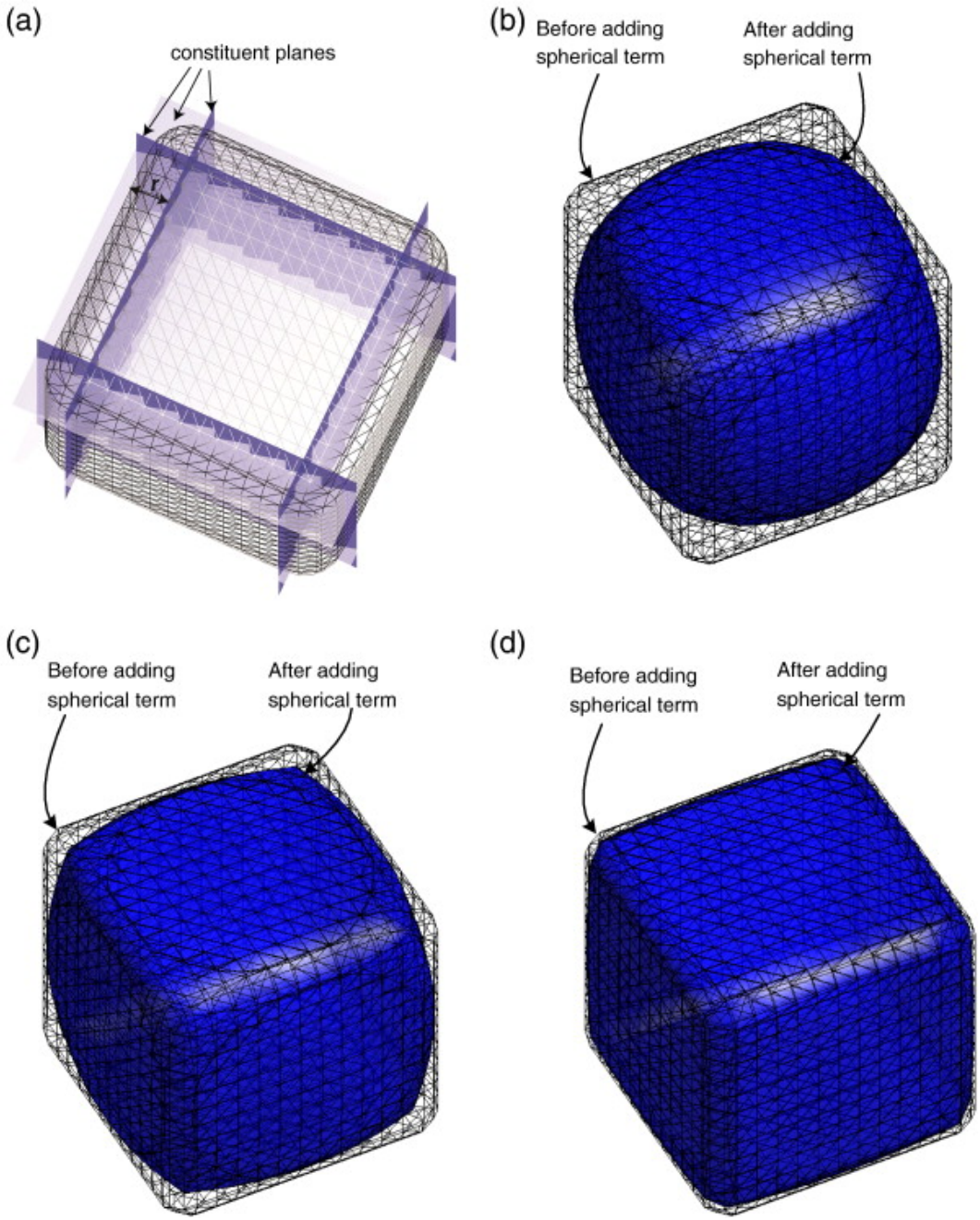


Fig. 6: Construction of potential particles (a) constituent planes are squared and expanded by a constant r. A fraction of sphere is added. Particles with the spherical term are visible in (b) k=0.9, (c) k=0.7, and (d) k=0.4 (after [Boon2013]).

where (a_i, b_i, c_i) is the unit normal vector of the ith plane, defined with respect to the particle's local coordinate system, and d_i is the distance of the plane to the local origin. According to [Boon2012], an inner, smooth potential particle is used to calculate the contact normal, formulated as in (4.32):

$$f = \sum_{i=1}^{N} \langle a_i x + b_i y + c_i z - d_i + r \rangle^2 \tag{4.32}$$

This potential particle is defined inner by a distance r inside the actual particle, with edges rounded by a radius or curvature r, as well (see Fig. *fig-pbInner*).

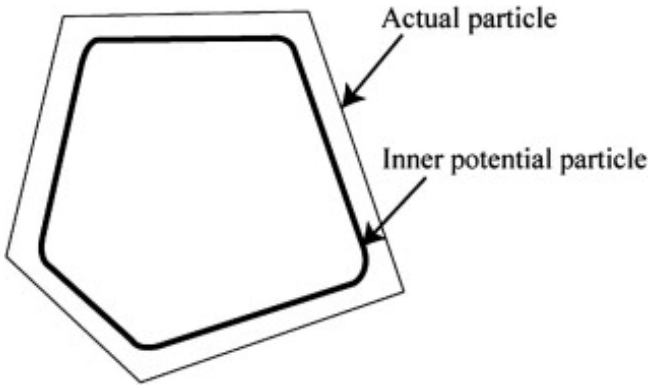


Fig. 7: A potential particle is defined inside the actual particle. The normal vector of the particle at any point can be calculated from the first derivative of the potential particle. (after [Boon2012]).

In YADE, the *Potential Blocks* have a slightly different mathematical expression, since their shape is generated as an assembly of planes as in (4.33):

$$a_i x + b_i y + c_i z - d_i - r = 0, i = 1 : N \tag{4.33}$$

while the inner *Potential Particle* used to calculate the contact normal is defined as in (4.34):

$$f = \sum_{i=1}^{N} \langle a_i x + b_i y + c_i z - d_i \rangle^2. \tag{4.34}$$

Now, the *Potential Block* surface is at a distance of $(d_i + r)$ from the local particle center, while the inner potential particle is at a distance d from the local particle center.

It is worth to emphasize on the fact that the shape of a *Potential Block* is defined using an assembly of planes and not a single, implicit potential function, like we have for the *Potential Particles* code. The inner potential particle in the *Potential Blocks* code is only used to calculate the contact normal.

The problem of establishing intersection between a pair of blocks is cast as a standard linear programming problem of finding a feasible region which satisfies all the linear inequalities defining both blocks. The contact point is calculated as the analytic centre of the feasible region, a well-known concept of interior-point methods in convex optimization calculations. The contact normal is obtained from the gradient of a smooth "potential particle" defined inside the block. The overlap distance is calculated through bi-section searching along the contact normal, within the overlap region.

The linear programming solver for *Potential Blocks* was originally CPLEX, but has been updated to CLP, developed by COIN-OR, since the latter can be downloaded from Ubuntu or Debian's distributions without requiring an academic licence.

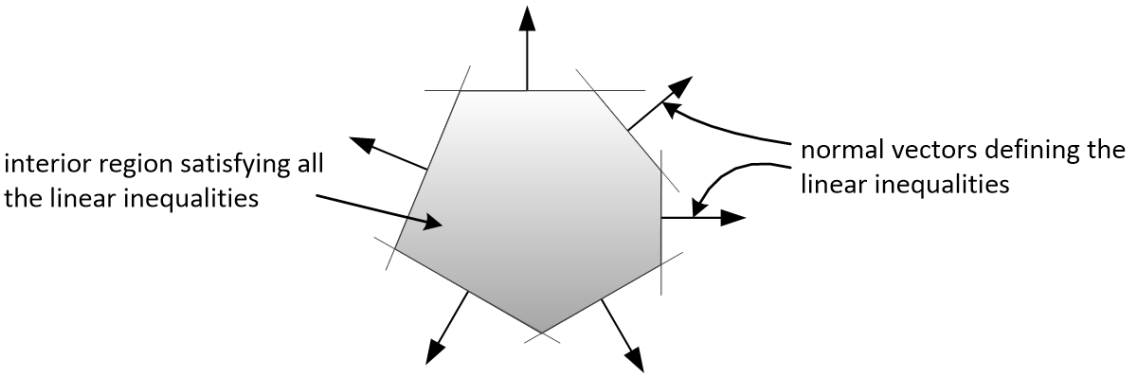


Fig. 8: A potential block. The normal vectors of the faces point outwards (after [Boon2013b]).

4.6.4 Engines

The PP and PB codes use their own classes to handle bounding volumes, contact geometry & physics and recording of outputs in vtk format, while they derive the interparticle friction angle from the frictional material class *FrictMat*. The syntax used to invoke these classes is similar, unless if specified otherwise.

Shape	*PotentialParticle*	*PotentialBlock*
Material	*FrictMat*	*FrictMat*
BoundFunctor	*PotentialParticle2AABB*	*PotentialBlock2AABB*
IGeom	*ScGeom*	*ScGeom*
IGeomFunctor	*Ig2_PP_PP_ScGeom*	*Ig2_PB_PB_ScGeom*
IPhys	*KnKsPhys*	*KnKsPBPhys*
IPhysFunctor	*Ip2_FrictMat_FrictMat_KnKsPhys*	*Ip2_FrictMat_FrictMat_KnKsPBPhys*
LawFunctor	*Law2_SCG_KnKsPhys_KnKsLaw*	*Law2_SCG_KnKsPBPhys_KnKsPBLaw*
VTK Recorder	*PotentialParticleVTKRecorder*	*PotentialBlockVTKRecorder*

A simple *simulation loop* using the *Potential Blocks* reads as:

```
O.engines=[
        ForceResetter(),
        InsertionSortCollider([PotentialBlock2AABB()], verletDist=0.01),
        InteractionLoop(
                [Ig2_PB_PB_ScGeom(twoDimension=True, unitWidth2D=1.0,␣
↪calContactArea=True)],
                [Ip2_FrictMat_FrictMat_KnKsPBPhys(kn_i=1e8, ks_i=1e7, Knormal=1e8,␣
↪Kshear=1e7, viscousDamping=0.2)],
                [Law2_SCG_KnKsPBPhys_KnKsPBLaw(label='law', neverErase=False,␣
↪allowViscousAttraction=False)]
        ),
        NewtonIntegrator(damping=0.2, exactAsphericalRot=True, gravity=[0,0,-9.81]),
        PotentialBlockVTKRecorder(fileName='./vtk/file_prefix', iterPeriod=1000,␣
↪twoDimension=True, sampleX=30, sampleY=30, sampleZ=30, maxDimension=0.2, label=
↪'vtkRecorder')
]
```

Attention should be given to the *twoDimension* parameter, which defines whether a contact should be handled as 2-D or 3-D.

4.6.5 Contact Law

In both codes, the normal force is calculated as:

$$\mathbf{F_n} = \mathtt{Knormal} \cdot A_c \cdot u_n \cdot \mathbf{n} \tag{4.35}$$

where $\mathtt{Knormal}$ the normal stiffness coefficient [kN/m^3]; A_c the contact area [m^2] and u_n the overlap distance. The normal stiffness of each contact [kN/m] is thus $k_n = \mathtt{Knormal} \cdot A_c$, where A_c is updated in every timestep.

The shear force is calculated incrementally, using a similar logic. The increment of the shear force vector before slipping of the contact is calculated as:

$$\Delta\mathbf{F_s} = -\mathtt{Kshear} \cdot A_c \cdot \Delta\mathbf{u_s} \tag{4.36}$$

where $\mathtt{Kshear}$ the shear stiffness coefficient [kN/m^3] and Δu_s the current relative shear displacement.

Contact Area

The contact area is calculated using a heuristic algorithm to detect points on the surface of the overlap volume, searching along the contact shear direction. In essence, it is calculated as the area of a 2D slice of the overlap volume along the shear direction, passing from the contact point. If `twoDimension=True`, the *contactArea* parameter is calculated as:

```
if(twoDimension) { phys->contactArea = phys->jointLength*unitWidth2D;}
```

The *unitWidth2D* parameter is defined by the user (usually equal to 1.0), denoting the out-of-plane width in 2-D simulations. The *contactArea* and *jointLength* parameters are calculated if *calContactArea* `=True`. In the opposite case, they are considered equal to 1.0 and the contact law is degenerated to a linear law with constant stiffness. A minimum value is considered for the *contactArea*, to represent cases where the overlap volume is practically a point.

Overlap distance

The overlap distance u_n is calculated using a bracketed bisection search algorithm along the contact normal direction, to find two opposite points on the surface of the overlap region, starting from the contact point. It is stored in the parameter *penetrationDepth*, as the distance between these two opposite points.

4.6.6 Shape definition of a PP and a PB

A strong merit of the *Potential Particles* and the *Potential Blocks* codes lies in the fact that the geometric definition of the particle shape and the contact detection problem are resolved using only the equations of the faces of the particles. In this way, using a single data structure, there is no need to store information about the vertices or their connectivity to establish contact, a feature that makes them computationally affordable, while all contacts are handled in the same way (there is no need to distinguish among face-face, face-edge, face-vertex, edge-edge, edge-vertex or vertex-vertex contacts). Due to this, the geometry of a particle is defined in the shape class using the values of the normal vectors of the faces and the distances of the faces from the local origin.

For example, to define a cuboid (6 faces) with rounded edges, an edge length of D, centred to its local centroid and aligned to its principal axes, using the *Potential Particles* code, we set:

```
r=D/10.
k=0.3
R=D/2.
```

(continues on next page)

(continued from previous page)

```
b=Body()
b.shape=PotentialParticle( r=r, k=k, R=R,
                           a=[   1.0,    -1.0,     0.0,     0.0,     0.0,     0.0],
                           b=[   0.0,     0.0,     1.0,    -1.0,     0.0,     0.0],
                           c=[   0.0,     0.0,     0.0,     0.0,     1.0,    -1.0],
                           d=[D/2.-r,  D/2.-r,  D/2.-r,  D/2.-r,  D/2.-r,  D/2.-r], ..
↪.)
```

The first element of the vector parameters a, b, c, d refers to the normal vector of the first plane and its distance from the local origin, the second element to the second plane, and so on.

Using the *Potential Particles* code, this is not a perfect cube, since the particle geometry is defined by a potential function as in (4.30). It is reminded that within this potential function, these planes are summed quadratically, the particle edges are rounded by a radius of curvature r and then the particle faces are curved by the addition of a "shadow" spherical particle with a radius R, to a percentage defined by the parameter k. A value r is deducted from each element of the vector parameter d, to compensate for expanding the potential particle by r.

The parameters a_i, b_i, c_i, d_i stated above correspond to the planes used in (4.33):

$$
\begin{aligned}
1.0x + 0.0y + 0.0z = D/2 &\Leftrightarrow +x = D/2 \\
-1.0x + 0.0y + 0.0z = D/2 &\Leftrightarrow -x = D/2 \\
0.0x + 1.0y + 0.0z = D/2 &\Leftrightarrow +y = D/2 \\
0.0x - 1.0y + 0.0z = D/2 &\Leftrightarrow -y = D/2 \\
0.0x + 0.0y + 1.0z = D/2 &\Leftrightarrow +z = D/2 \\
0.0x + 0.0y - 1.0z = D/2 &\Leftrightarrow -z = D/2
\end{aligned}
$$

To model a cube with an edge of D, using the *Potential Blocks* code, we define:

```
r=D/10.
R=D/2.*sqrt(3)
b=Body()
b.shape=PotentialBlock( r=r, R=R,
                        a=[   1.0,    -1.0,     0.0,     0.0,     0.0,     0.0],
                        b=[   0.0,     0.0,     1.0,    -1.0,     0.0,     0.0],
                        c=[   0.0,     0.0,     0.0,     0.0,     1.0,    -1.0],
                        d=[D/2.-r,  D/2.-r,  D/2.-r,  D/2.-r,  D/2.-r,  D/2.-r], ...)
```

Using the *Potential Blocks* code, this particle will have sharp edges and flat faces in what regards its geometry (i.e. the space it occupies), defined by the given planes, while for the calculation of the contact normal, an inner potential particle with rounded edges is used, formulated as in (4.34), located fully inside the actual particle. The distances of the planes from the local origin, stored in the vector parameter d, are reduced by r to achieve an exact edge length of D, using (4.33). The value of r must be sufficiently small, so that $d_{min} - r > 0$, while it should be sufficiently large, to allow for a proper calculation of the gradient of the inner Potential Particle at the contact point. A recommended value is $r \approx 0.5 * d_{min}$.

To ensure numerical stability, it is advised to normalize the normal vector of each plane, so that ${a_i}^2 + {b_i}^2 + {c_i}^2 = 1$. There is no limit to the number of the particle faces that can be used, a feature that allows the modelling of a variety of convex particle shapes.

In practice, it is usual for the geometry of a particle to be given in terms of vertices & their connectivity (e.g. in the form of a surface mesh, like in .stl files). In such cases, the user can calculate the normal vector of each face, which will give the coefficients a_i, b_i, c_i and using a vertex of each face, then calculate the coefficients d_i. A python routine to perform this without any additional effort by the user is currently being developed.

4.6.7 Body definition of a PP and a PB

To define a body using the *PotentialParticle* or *PotentialBlock* shape classes, it has to be assembled using the `_commonBodySetup` function, which can be found in the file py/utils.py. For example, to define a *PotentialParticle*:

```
O.materials.append(FrictMat(young=-1,poisson=-1,
↪frictionAngle=radians(0.0),density=2650,label='frictionless'))

b=Body()
b.shape=PotentialParticle(...)
b.aspherical=True # To be used in conjunction with exactAsphericalRot=True in the␣
↪NewtonIntegrator
# V: Volume
# I11, I22, I33: Principal inertias
utils._commonBodySetup(b,V,Vector3(I11,I22,I33), material='frictionless', pos=(0,0,
↪0), fixed=False)
b.state.pos=Vector3(xPos,yPos,zPos)
b.state.ori=Quaternion((random.random(),random.random(),random.random()),random.
↪random())
b.shape.volume=V;
O.bodies.append(b)
```

The *PotentialParticle* must be initially defined, so that the local axes coincide with its principal axes, for which the inertia tensor is diagonal. More specifically, the plane coefficients (a_i, b_i, c_i) defining the plane normals must be rotated, so that when the orientation of the particle is zero, the *PotentialParticle* is oriented to its principal axes.

It should be noted that the principal inertia values `I11, I22, I33` mentioned here are divided with the density of the considered material, since they are multiplied with the density inside the `_commonBodySetup` function. The mass of the particle is calculated within the same function as well, so we do not need to set manually `b.mass=V*density`.

For the *Potential Particles*, the volume and inertia must be calculated manually and assigned to the body as demonstrated above. For the *Potential Blocks*, an automatic calculation has been implemented for the volume and inertia tensor, the user does not have to define the particle to its principal axes, since this is handled automatically within the source code, while if no value is given for the parameter R, it is calculated as half the distance of the farthest vertices.

For example, to define a *PotentialBlock*:

```
O.materials.append(FrictMat(young=-1,poisson=-1,
↪frictionAngle=radians(0.0),density=2650,label='frictionless'))

b=Body()
b.shape=PotentialBlock(R=0.0, ...) #here we set R=0.0 to trigger automatic␣
↪calculation of R
b.aspherical=True # To be used in conjunction with exactAsphericalRot=True
utils._commonBodySetup(b,b.shape.volume,b.shape.inertia, material='frictionless',␣
↪pos=Vector3(xPos,yPos,zPos), fixed=False)
b.state.ori=b.shape.orientation # this will rotate the particle to its initial random␣
↪system. If b.state.ori=Quaternion.Identity, the PB is oriented to its principal axes
O.bodies.append(b)
```

4.6.8 Boundary Particles

The PP & PB codes support the definition of *boundary* particles, which interact only with *non-boundary* ones. These particles can have a variety of uses, e.g. to model loading plates acting on a granular sample, while different uses can emerge for different applications. A particle can be set as a boundary one in both codes, using the boolean parameter *isBoundary* inside the shape class.

In the PP code, all particles interact with the same normal and shear contact stiffness *Knormal* and *Kshear*, defined in the *Ip2_FrictMat_FrictMat_KnKsPhys* functor.

The PB code supports the definition of different contact stiffness values for interactions between *boundary* and *non-boundary* or *non-boundary* and *non-boundary* particles. When `isBoundary=False`, the *PotentialBlock* in question is handled to interact with normal and shear stiffness coefficients *Knormal* and *Kshear*, respectively, with other non-boundary particles. When `isBoundary=True`, the *PotentialBlock* in question is handled to interact with normal and shear stiffness coefficients *kn_i* and *ks_i*, respectively, with non-boundary particles.

4.6.9 Visualization

Visualization of the *PotentialParticle* and *PotentialBlock* shape classes is offered using the qt environment (OpenGL). Additionally, the *export.VTKExporter.exportPotentialBlocks* function and *PotentialParticleVTKRecorder* and *PotentialBlockVTKRecorder* engines can be used to export geometrical and interaction information of the analyses in vtk format (visualized in Paraview). It should be noted that currently the *PotentialBlockVTKRecorder* records a rounded approximation of the particle, rather than the actual particle with sharp corners and edges.

In the qt environment, the *PotentialParticle* shape class is visualized using the Marching Cubes algorithm, and the level of display accuracy can be determined by the user. This is controlled by the parameters:

```
# Potential Particles
Gl1_PotentialParticle.sizeX=20
Gl1_PotentialParticle.sizeY=20
Gl1_PotentialParticle.sizeZ=20
```

A similar choice exists for output in vtk format, using the *PotentialParticleVTKRecorder* or *PotentialBlockVTKRecorder*, syntaxed as:

```
# Potential Particles
PotentialParticleVTKRecorder(sampleX=30, sampleY=30, sampleZ=30, maxDimension=20)

# Potential Blocks
PotentialBlockVTKRecorder(sampleX=30, sampleY=30, sampleZ=30, maxDimension=20)
```

The parameters sizeX,Y,Z (for OpenGL visualization) and sampleX,Y,Z (for output in vtk format) represent the number of subdivisions of the Aabb of the particle to a grid, which will be used to draw its geometry, in respect to the global axes X, Y, Z. Larger values will result to a more accurate display of the particles' shape, but will slow down the visualization speed in qt and writing speed of the .vtk files and increase the size of the .vtk files. For output in vtk format, users can also define the parameter *maxDimension*, which overrides the selected sampleX,Y,Z values if they are too small, as described below:

$$\text{if } |xmax - xmin|/sampleX > maxDimension \Rightarrow sampleX = |xmax - xmin|/maxDimension$$
$$\text{if } |ymax - ymin|/sampleY > maxDimension \Rightarrow sampleY = |ymax - ymin|/maxDimension$$
$$\text{if } |zmax - zmin|/sampleZ > maxDimension \Rightarrow sampleZ = |zmax - zmin|/maxDimension$$

The *PotentialParticleVTKRecorder* and *PotentialBlockVTKRecorder* also support optionally the recording of the particles' velocities (linear and angular), interaction information (contact point and forces), colors and ids, using:

```
# Potential Particles
PotentialParticleVTKRecorder(..., REC_VELOCITY=True, REC_INTERACTION=True,␣
↪REC_COLORS=True, REC_ID=True)

# Potential Blocks
PotentialBlockVTKRecorder(..., REC_VELOCITY=True, REC_INTERACTION=True,␣
↪REC_COLORS=True, REC_ID=True)
```

Force chains and other visual outputs are available in qt by default, while they can be extracted in vtk format using the classic *VTKRecorder* or the *export.VTKExporter* class.

A boolean parameter *twoDimension* exists to specify whether the particles will be rendered as 2-D or 3-D in the vtk output:

```
# Potential Particles
PotentialParticleVTKRecorder(..., twoDimension=False)

# Potential Blocks
PotentialBlockVTKRecorder(..., twoDimension=False)
```

This parameter should not be mixed up with the *Ip2_FrictMat_FrictMat_KnKsPBPhys.twoDimension* parameter, which is used to define how the contact forces are calculated, as described in the *Engines* section.

4.6.10 Axis-Aligned Bounding Box

The PP & PB codes use their own BoundFunctors, called *PotentialParticle2AABB* and *PotentialBlock2AABB*, respectively, to define the Axis-Aligned Bounding Box of each particle. In both bound functors, a boolean parameter *AabbMinMax* exists, allowing the user to choose between an approximate cubic Aabb or a more accurate one.

In particular, if `AabbMinMax=False`, a cubic Aabb is considered with dimensions `1.0*R`. This is implemented for both the PP and PB codes, even though the *Potential Blocks* do not have a spherical term. In this case, the radius R is used as a reference length, denoting half the diagonal of the cubic Aabb. Usage of this approximate cubic Aabb is not advised in general, since it can increase the number of empty contacts, adding thus to the time needed to facilitate the approximate contact detection, while it relies on the radius R, the value of which should enclose the whole particle if this option is activated.

If `AabbMinMax=True`, a more accurate Aabb can be defined. Currently, the initial Aabb of a *PotentialParticle* has to be defined manually by the user, in the particle local coordinate system and for the initial orientation of the particle. To do so, the user has to manually specify the two extreme points of the Aabb: *minAabbRotated*, *maxAabbRotated* inside the shape class. The Aabb for a *PotentialBlock*, on the other hand, is calculated and updated automatically from the vertices of the particle, if the boolean parameter *AabbMinMax* `=True`.

As discussed in the subsection *Visualization*, the dimensions of the Aabb are used as a drawing space in the code implementing rendering of the particles in the qt environment (for the PP code) and for the creation of the output files in vtk format (for both codes). This is achieved by using two auxiliary parameters: *minAabb* and *maxAabb*. For the *Potential Blocks* code only, if these parameters are left unassigned, the drawing space is configured automatically inside the *PotentialBlockVTKRecorder* using the Aabb of the particle. For the particles to be properly rendered as closed surfaces in both qt and vtk outputs using the available codes, we need to define a drawing space slightly larger than the actual one. Here, this drawing space is represented by the Aabb of the particles, and thus the differentiation between the minAabb, maxAabb and minAabbRotated, maxAabbRotated stems from the need to satisfy two conditions: 1. The Aabb used for primary contact detection must be as tight as possible, in order to have the least number of empty contacts and 2. The Aabb used as a rendering space must be slightly larger, in order to have proper rendering. If a dimension of the Aabb used for visualization purposes is defined smaller than the actual one, the faces on that side of the particle are rendered as hollow and only the edges are visualised, a functionality that can be used to e.g. see through boundaries, like demonstrated in the vtk output of the examples/PotentialParticles/cubePPscaled.py example.

To recap, in the *Potential Particles* code, the *minAabbRotated* and *maxAabbRotated* parameters define the initial Aabb used to facilitate primary contact detection, while the *minAabb* and *maxAabb* parameters are used for visualization of the particles in qt and vtk outputs. In the *Potential Blocks* code, the Aabb used to facilitate primary contact detection is calculated automatically from the particles' vertices, which are also used for visualization in qt, while the parameters *minAabb* and *maxAabb* are used for visualization in vtk outputs and can be left unassigned, to trigger an automatic configuration of the drawing space of the particle in the *PotentialBlockVTKRecorder*.

Two brief examples demonstrating the syntax of these features can be found below.

For the *Potential Particles* code:

```
b=Body()
b.shape=PotentialParticle(AabbMinMax=True,
                          minAabbRotated=Vector3(xmin,ymin,zmin),
                          maxAabbRotated=Vector3(xmax,ymax,zmax),
                          minAabb=Vector3(xmin,ymin,zmin),
                          maxAabb=Vector3(xmax,ymax,zmax), ...)
```

For the *Potential Blocks* code:

```
b=Body()
b.shape=PotentialBlock(AabbMinMax=True,
                       minAabb=Vector3(xmin,ymin,zmin),
                       maxAabb=Vector3(xmax,ymax,zmax), ...)
```

4.6.11 Block Generation algorithm

The *Potential Blocks* code is compatible with the *Block Generation* algorithm introduced in [Boon2015], which can split particles by their intersection with discontinuity planes, initially developed for the study of rock-masses. This code is hardcoded in YADE in the form of a Preprocessor. Using a single data structure for the definition of the particle shape and the definition of the discontinuities, as well, allows the generation of a large number of particles at a reasonable computational cost. The sequential subdivision concept is used along with a linear programming framework. Non-persistent joints can be modelled by introducing more constraints.

An example to demonstrate the usage of this code exists in examples/PotentialBlocks/WedgeYADE.py The discontinuity planes used in this script are included in a csv format in examples/PotentialBlocks/joints/jointC.csv.

The documentation on how to use this code is currently being written.

4.6.12 Examples

Examples can be found in the folders examples/PotentialParticles and examples/PotentialBlocks/, where the syntax of the codes is demonstrated.

4.6.13 Disclaimer

These codes were developed for academic purposes. Some variables are no longer in use, as the PhD thesis of the original developer spanned over many years, with numerous trials and errors. As this piece of code has many dependencies within the YADE ecosystem, user discretion is advised.

4.6.14 References

To acknowledge our scientific contribution, please cite the following:

Potential Blocks

- Boon CW (2013) Distinct Element Modelling of Jointed Rock Masses: Algorithms and Their Verification. D.Phil. Thesis, University of Oxford
- Boon CW, Houlsby GT, Utili S (2012) A new algorithm for contact detection between convex polygonal and polyhedral particles in the discrete element method. Computers and Geotechnics, 44: 73-82

Potential Particles

- Houlsby GT (2009) Potential particles: a method for modelling non-circular particles in DEM. Computers and Geotechnics, 36(6):953-959
- Boon CW, Houlsby GT, Utili S (2013) A new contact detection algorithm for three dimensional non-spherical particles. Powder Technology, S.I. on DEM, 248: 94-102

Block Generation

- Boon CW, Houlsby GT, Utili S (2015) A new rock slicing method based on linear programming. Computers and Geotechnics, 65:12-29

4.7 Bayesian Calibration using GrainLearning

Bayesian calibration is a probabilistic method for estimating the parameters of a computer model. The output of Bayesian Calibration are conditional probability distributions of model parameters **conditioned on** (experimental or theoretical) reference data. Here, we will apply Bayesian calibration to DEM models of granular materials, using the GrainLearning package. Check out the GrainLearning documentation for more information. Essentially, what a Yade user needs to do is to define a *callback function* that runs their Yade script , e.g., in batch mode, with a parameter table given by GrainLearning. GrainLearning updates this table iteratively, based on the provided reference data, until the error tolerance is met.

4.7.1 Installation

Stable versions of GrainLearning can be installed via *pip install grainlearning*. However, you would still need to clone the GrainLearning repository to run the tutorials.

```
# create a virtual environment
python3 -m venv env
source env/bin/activate

# install GrainLearning
pip install grainlearning[visuals]

# Clone the repository (optional)
git clone https://github.com/GrainLearning/grainLearning.git

# run a simple linear regression test (optional)
python3␣
↪grainLearning/tutorials/simple_regression/linear_regression/python_linear_regression_solve.py

# deactivate virtual environment (optional)
deactivate
rm -r env
```

Alternatively, you can include other optional modules of GrainLearning by passing the corresponding extras to *pip install*, such as *pip install grainlearning[dev]* or *pip install grainlearning[rnn]* or simply *pip install grainlearning[all]*.

Background

In Bayesian filtering (updating the probability distribution of a model state given data sequences), deterministic models are made stochastic by adding unknown modeling and observation errors. We typically refer to the systems that these stochastic models describe as **dynamic systems**. The **state** of a dynamic system is the set of random variables that describe the system at a given time.

4.7.2 Dynamic Systems

The *dynamic_systems* module of GrainLearning encapsulates simulation and reference data in a single *DynamicSystem* class. The *IODynamicSystem* class sends instructions to external *third-party software* like Yade and retrieves simulation data from the output files of the software.

Note

A dynamic system is also known as a state-space model in the literature. It describes the time evolution of the state of the model $\mathbf{x}_t$ and the state of the observables $\mathbf{y}_t$. Both $\mathbf{x}_t$ and $\mathbf{y}_t$ are random variables whose distributions are updated by the *inference* module.

$$\mathbf{x}_t = \mathbb{F}(\mathbf{x}_{t-1}) + \boldsymbol{\nu}_t$$
$$\mathbf{y}_t = \mathbb{H}(\mathbf{x}_t) + \boldsymbol{\omega}_t$$

where $\mathbb{F}$ represents the **third-party software** model that takes the previous model state $\mathbf{x}_{t-1}$ to make predictions for time t. If all observables $\mathbf{y}_t$ are independent and have a one-to-one correspondence with $\mathbf{x}_t$, (meaning you predict what you observe), the observation model $\mathbb{H}$ reduces to the identity matrix $\mathbb{I}_d$, with d being the number of independent observables.

The simulation and observation errors $\boldsymbol{\nu}_t$ and $\boldsymbol{\omega}_t$ are random variables and assumed to be normally distributed around zero means. We consider both errors together in the covariance matrix. For more information on the *dynamic_systems* module, see GrainLearning documentation

4.7.3 Bayesian Filtering

Bayesian filtering is a general framework for estimating the *hidden* state of a dynamical system from partial observations using a predictive model of the system dynamics.

The state, usually augmented by the system's parameters, changes in time according to a stochastic process, and the observations are assumed to contain random noise. The goal of Bayesian filtering is to update the probability distribution of the system's state whenever new observations become available, using the recursive Bayes' theorem.

Humans are Bayesian machines, constantly using Bayesian reasoning to make decisions and predictions about the world around them. Bayes' theorem is the mathematical foundation for this process, allowing us to update our beliefs in the face of new evidence,

$$p(A|B) = \frac{p(B|A)p(A)}{p(B)}.$$

Note

- $p(A|B)$ is the **posterior** probability of hypothesis A given evidence B has been observed
- $p(B|A)$ is the **likelihood** of observing evidence B given hypothesis A
- $p(A)$ is the **prior** probability of hypothesis A
- $p(B)$ is a **normalizing** constant that ensures the posterior distribution sums to one

At its core, Bayes' theorem is a simple concept: the probability of a hypothesis given some observed evidence is proportional to the product of the prior probability of the hypothesis and the likelihood of the evidence given the hypothesis.

The method currently available for statistical inference is *Sequential Monte Carlo*. It recursively updates the probability distribution of the augmented model state $\hat{\mathbf{x}}_T = (\mathbf{x}_T, \boldsymbol{\Theta})$ from the sequences of observation data $\mathbf{y}_{0:T}$ from time $t = 0$ to T. The posterior distribution of the augmented model state is approximated by a set of samples, where each sample instantiates a realization of the model state.

Via Bayes' rule, the posterior distribution of the *augmented model state* reads

$$p(\hat{\mathbf{x}}_{0:T}|\mathbf{y}_{1:T}) \propto \prod_{t_i=1}^{T} p(\mathbf{y}_{t_i}|\hat{\mathbf{x}}_{t_i}) p(\hat{\mathbf{x}}_{t_i}|\hat{\mathbf{x}}_{t_i-1}) p(\hat{\mathbf{x}}_0),$$

Where $p(\hat{x}_0)$ is the initial distribution of the model state. We can rewrite this equation in the recursive form, so that the posterior distribution gets updated at every time step t.

$$p(\hat{x}_{0:t}|y_{1:t}) \propto p(y_t|\hat{x}_t)p(\hat{x}_t|\hat{x}_{t-1})p(\hat{x}_{1:t-1}|y_{1:t-1}),$$

Where $p(y_t|\hat{x}_t)$ and $p(\hat{x}_t|\hat{x}_{t-1})$ are the likelihood distribution and the transition distribution, respectively. The likelihood distribution is the probability distribution of observing y_t given the model state $\hat{x}_t$. The transition distribution is the probability distribution of the model's current state $\hat{x}_t$ given its previous state $\hat{x}_{t-1}$.

Note

We apply no perturbation in the parameters Θ nor in the model states $x_{1:T}$ because the model history must be kept intact for path-dependent materials. This results in a deterministic transition distribution predetermined from the initial state $p(\hat{x}_0)$.

The prior, likelihood, and posterior distributions can be evaluated via importance sampling. The idea is to have samples that are more important than others when approximating a target distribution. The measure of this importance is the so-called **importance weight** (see the figure below).

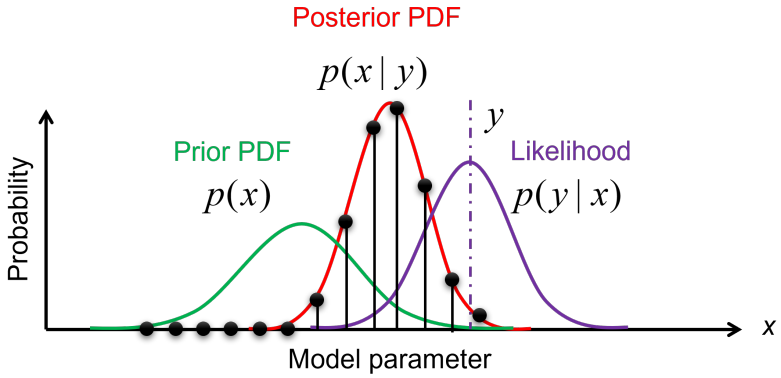


Fig. 9: Illustration of importance sampling.

Therefore, we draw samples, $\Theta^{(i)}$ $(i = 1, ..., N_p)$, from a proposal density, leading to an ensemble of the model state $x_t^{(i)}$. The importance weights $w_t^{(i)}$ are updated recursively, via

$$w_t^{(i)} \propto p(y_t|\hat{x}_t^{(i)})p(\hat{x}_t^{(i)}|\hat{x}_{t-1}^{(i)})w_{t-1}^{(i)}.$$

The likelihood $p(y_t|\hat{x}_t^{(i)})$ can be assumed to be a multivariate Gaussian (see the equation below), which is computed by the function *get_likelihoods* of the "Sequential Monte Carlo (SMC)" class.

$$p(y_t|\hat{x}_t^{(i)}) \propto \exp\{-\frac{1}{2}[y_t - H(x_t^{(i)})]^T \Sigma_t^{D\,-1} [y_t - H(x_t^{(i)})]\},$$

where H is the observation model that reduces to a diagonal matrix for uncorrelated observables, and Σ_t^D is the covariance matrix calculated by multiplying y_t along the diagonal and the user-defined normalized variance *sigma_max*.

By making use of importance sampling, the posterior distribution $p(y_t|\hat{x}_t^{(i)})$ gets updated over time — this is known as Bayesian updating. Figure below illustrates the evolution of a posterior distribution over time.

Since the importance weight on each sample $\Theta^{(i)}$ is discrete and the sample $\Theta^{(i)}$ and model state $x_t^{(i)}$ have one-to-one correspondence, the ensemble mean and variance of f_t, an arbitrary function of the

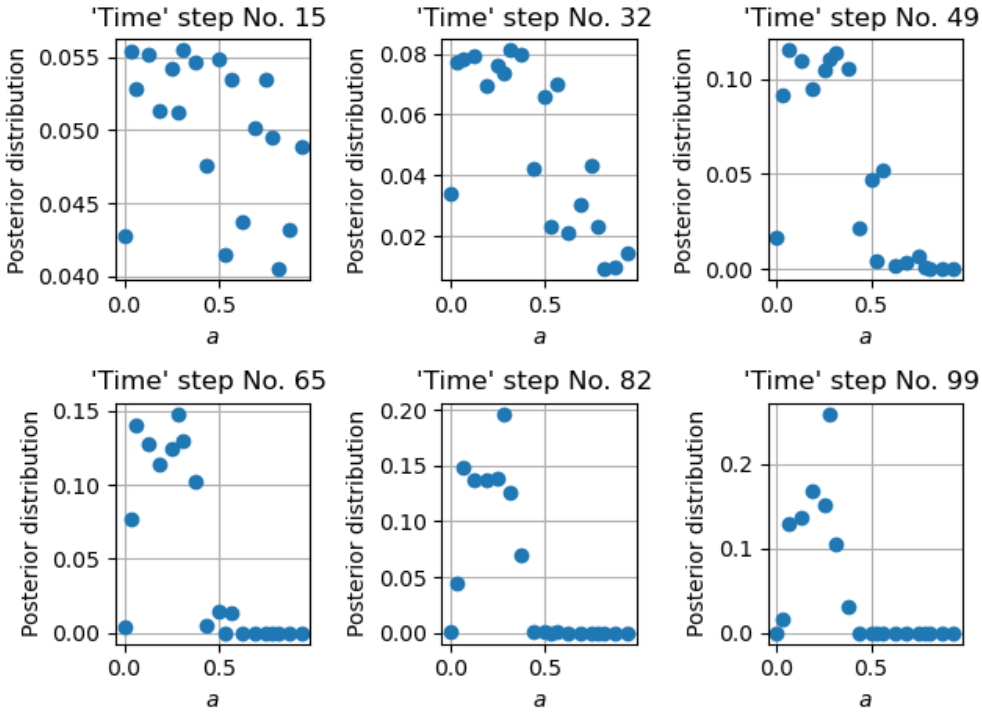


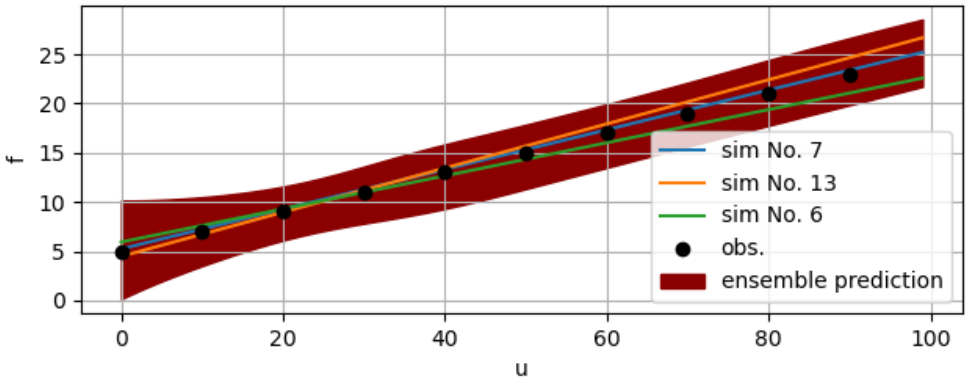
Fig. 10: Time evolution of the importance weights over model parameter $\mathfrak{a}$.

model's augmented state $\hat{\mathbf{x}}_t$, can be computed as

$$\widehat{\mathrm{E}}[f_t(\hat{\mathbf{x}}_t)|\mathbf{y}_{1:t}] = \sum_{i=1}^{N_p} w_t^{(i)} f_t(\hat{\mathbf{x}}_t^{(i)}),$$

$$\widehat{\mathrm{Var}}[f_t(\hat{\mathbf{x}}_t)|\mathbf{y}_{1:t}] = \sum_{i=1}^{N_p} w_t^{(i)} (f_t(\hat{\mathbf{x}}_t^{(i)}) - \widehat{\mathrm{E}}[f_t(\hat{\mathbf{x}}_t)|\mathbf{y}_{1:t}])^2,$$

The figure below gives an example of the ensemble prediction in darkred, the top three fits in blue, orange, and green, and the observation data in black.

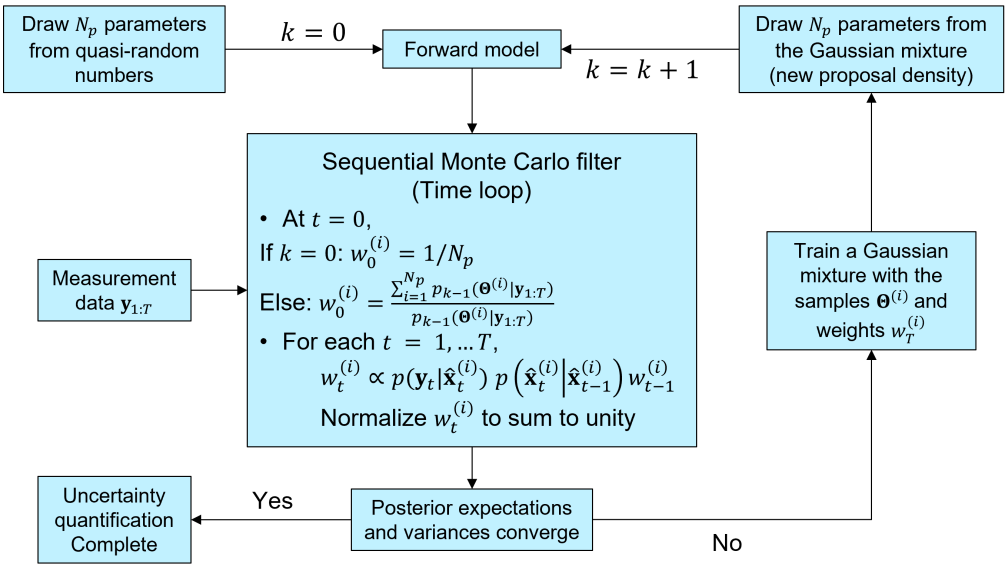


The idea of iterative Bayesian filtering algorithm is to solve the inverse problem all over again, with new samples drawn from a more sensible proposal density, leading to a multi-level resampling strategy to avoid weight degeneracy and improve efficiency. The essential steps include

1. Generating the initial samples using a low-discrepancy sequence (Halton, Sobol, or Latin hypercube sampling),

2. Running the instances of the predictive model via a user-defined *callback function*,
3. Estimating the time evolution of the posterior distribution,
4. Generating new samples from the proposal density, trained with the previous ensemble (i.e., samples and associated weights),
5. Check whether one of the stopping criteria (GrainLearning ensemble error, individual sample error, or non-dimensional variance < tolerance) is met, and stop the iteration if so.
6. If not, repeating step 1–5.

The figure below illustrates the workflow of iterative Bayesian filtering.



More details on the iterative Bayesian filtering algorithm can be found in the following papers.

- H. Cheng, T. Shuku, K. Thoeni, P. Tempone, S. Luding, V. Magnanimo, (2019) An iterative Bayesian filtering framework for fast and automated calibration of DEM models. *Computer Methods in Applied Mechanics and Engineering, 350*, pp. 268-294, 10.1016/j.cma.2019.01.027
- P. Hartmann, H. Cheng, K. Thoeni, (2022) Performance study of iterative Bayesian filtering to develop an efficient calibration framework for DEM. *Computers and Geotechnics, 141*, 104491, 10.1016/j.compgeo.2021.104491
- H. Cheng, L. Orozco, R. Lubbe, A. Jansen, P. Hartmann, K. Thoeni, (2024). GrainLearning: A Bayesian uncertainty quantification toolbox for discrete and continuum numerical models of granular materials. *Journal of Open Source Software, 9(97)*, 6338, 10.21105/joss.06338

Setting up a case

4.7.4 In Yade

Modification to your Yade script is very minimal. First, we need to add a "description" field to the tags of Yade so that each simulation can be uniquely identified.

```
# check if run in batch mode
isBatch = runningInBatch()
if isBatch:
    description = O.tags['description']
else:
    description = 'test_run'
```

The *plot* module of Yade saves simulation data into a dictionary via *plot.plots* and *plot.addData*.

```
data_file_name = f'{description}_sim.txt'
data_param_name = f'{description}_param.txt'
def add_sim_data():
    inter = O.interactions[0, 1]
    plot.addData(u=inter.geom.penetrationDepth, f=inter.phys.normalForce.norm())
```

In addition to these data which will be compared to the reference data in order to calculate probabilities, the corresponding parameter values used by Yade also need to be stored. This can be achieved with the *write_dict_to_file* helper function of *GrainLearning*.

```
# initialize data dictionary
param_data = {}
for name in table.__all__:
    param_data[name] = eval('table.' + name)
# write simulation data into a text file
write_dict_to_file(plot.data, data_file_name)
write_dict_to_file(param_data, data_param_name)
```

That's everything you need to do to make your DEM simulation ready for a Bayesian calibration. Download `this script` to check see to set up Bayesian calibration for particle-particle collisions.

4.7.5 In GrainLearning

As a Python package, GrainLearning can be imported into your Python script as follows.

```
from grainlearning import BayesianCalibration
from grainlearning.dynamic_systems import IODynamicSystem
```

To be able to run Yade from the *callback function*, you need to specify the path to the Yade executable and the Yade script.

```
PATH = os.path.abspath(os.path.dirname(__file__))
executable = 'yade-batch'
yade_script = f'{PATH}/Collision.py'
```

Because Yade can take parameter values conveniently from a text file, in the batch mode, we only need the following line where an updated parameter data file is passed to *yade-batch*.

```
def run_sim(calib):
    """
    Run the external executable and passes the parameter sample to generate the␣
↪output file.
    """
    print("*** Running external software YADE ... ***\n")
    os.system(' '.join([executable, calib.system.param_data_file, yade_script]))
```

Alternatively, you can run Yade from shell scripts through the *run_yade_from_shell* function of grainlearning.tools. The folder */examples/Bayesian_calibration/platform_shells/* contains predefined shell scripts for various platforms, including `desktop`, `HPC cluster`, and `AWS cloud`.

```
path_to_shell = 'platform_shells/desktop/'
def run_sim(calib):
    """
    Run the external executable and passes the parameter sample to generate the␣
↪output file.
    """
    print("*** Running external software YADE ... ***\n")
```

(continues on next page)

(continued from previous page)

```
    run_yade_from_shell(calib.system.param_data_file, yade_script, path_to_shell,␣
↪platform='desktop')
```

Obviously, one has to determine the number of unknown parameters before configuring further the Bayesian calibration problem. In one of our previous papers (Hartmann et al., 2022), we have shown that a number of parameters between $5 * N \log N$ and $10 * N \log N$, where N is the number of unknown parameters, is a good choice for the calibration of DEM models. Considering the example of two particle collision, the parameters could be the Young's modulus E_m and the Poisson's ratio ν.

```
param_names = ['E_m', 'nu']
num_samples = int(5 * len(param_names) * log(len(param_names)))
```

The *BayesianCalibration* class is initialized with a dictionary that contains all the necessary information for the calibration. The most important settings are the number of iterations, the error tolerance, the callback function, the upper and lower bounds of the parameters, the number of samples per iteration, and the normalized covariance tolerance (optional). See the example below.

```
# define the Bayesian Calibration object
calibration = BayesianCalibration.from_dict(
    {
        # maximum number of GL iterations
        "num_iter": 5,
        # error tolerance to stop the calibration
        "error_tol": 0.1,
        # call back function to run YADE
        "callback": run_sim,
        # DEM model as a dynamic system
        "system": {
            "system_type": IODynamicSystem,
            "param_min": [7, 0.0],
            "param_max": [11, 0.5],
            "param_names": param_names,
            "num_samples": 10,
            "obs_data_file": PATH + '/collision_obs.dat',
            "obs_names": ['f'],
            "ctrl_name": 'u',
            "sim_name": 'collision',
            "sim_data_dir": PATH + '/sim_data/',
            "sim_data_file_ext": '.txt',
            "sigma_tol": 0.01,
        },
        "inference": {
            "Bayes_filter": {
                # scale the covariance matrix with the maximum observation data or not
                "scale_cov_with_max": True,
            },
            "sampling": {
                # maximum number of distribution components
                "max_num_components": 1,
                # fix the seed for reproducibility
                "random_state": 0,
                # use slice sampling (set to False if faster convergence is designed.␣
↪However, the results could be biased)
                "slice_sampling": True,
            }
        },
```

(continues on next page)

(continued from previous page)

```
        # flag to save the figures (-1: no, 0: yes but only show the figures , 1: yes)
        "save_fig": 0,
        # number of threads to be used per simulation
        "threads": 1,
    }
)
```

If you want to assume modeling and observation error increases with their actual values, you can set the *scale_cov_with_max* to *False*. If the parameter distributions are multi-modal, the *max_num_components* sets the maximum number of components in the Gaussian mixture model. Since a variational version of the Gaussian mixture is used, the algorithm will tend to reduce the number of components, avoiding overfitting. Lastly, instead of directly sample from the Gaussian mixture, we can use low-discrepancy sequences to draw samples within a volume bounded by certain cutoff values on the probability density. This ensures that the Bayesian filter is unbiased. However, the convergence might be slower.

4.7.6 Running Bayesian calibration

The calibration is started by calling the *run* method of the *BayesianCalibration* object.

```
calibration.run()
```

If you have completed your simulation outside GrainLearning and simply want to check the statistics and generate a new parameter table,

```
calibration.load_and_run_one_iteration()
```

If you exit the calibration accidentally, you can resume it by loading all existing simulations to run the inference for one iteration, before calling the *run* method again.

```
calibration.load_and_run_one_iteration()
# when resuming the calibration, the current iteration number must be increased
calibration.increase_curr_iter()
calibration.run()
```

4.7.7 Setting the stopping criteria

There are three criteria GrainLearning checks to stop the iterative Bayesian Calibration process. The calibration stops if one of the following conditions is met: - The current normalized variance *sigma_max* has decrease below its tolerance value *sigma_tol* - The mean absolute percentage error of the ensemble prediction is below its tolerance value *gl_error_tol* - The mean absolute percentage error of the individual sample is below its tolerance value *error_tol*

```
# define the Bayesian Calibration object
calibration.error_tol = 0.01
calibration.gl_error_tol = 0.01
calibration.system.sigma_tol = 0.01
```

4.7.8 Analyzing and visualizing the results

Most of the time, a user is interested in the most probable parameter values. This can be obtained by calling the *get_most_prob_params* method of the *BayesianCalibration* object. GrainLearning also provides a set of plotting functions to visualize the results. For example, you can call *plot_uq_in_time* to show the parameter distributions and the comparison between the observation and the top three most probable simulations. Setting the verbose flag to true will give all the detailed statistics, including the "time" evolution of importance weights and the means and coefficients of variation of the parameters.

```
calibration.plot_uq_in_time()
calibration.plot_uq_in_time(verbose=True)
```

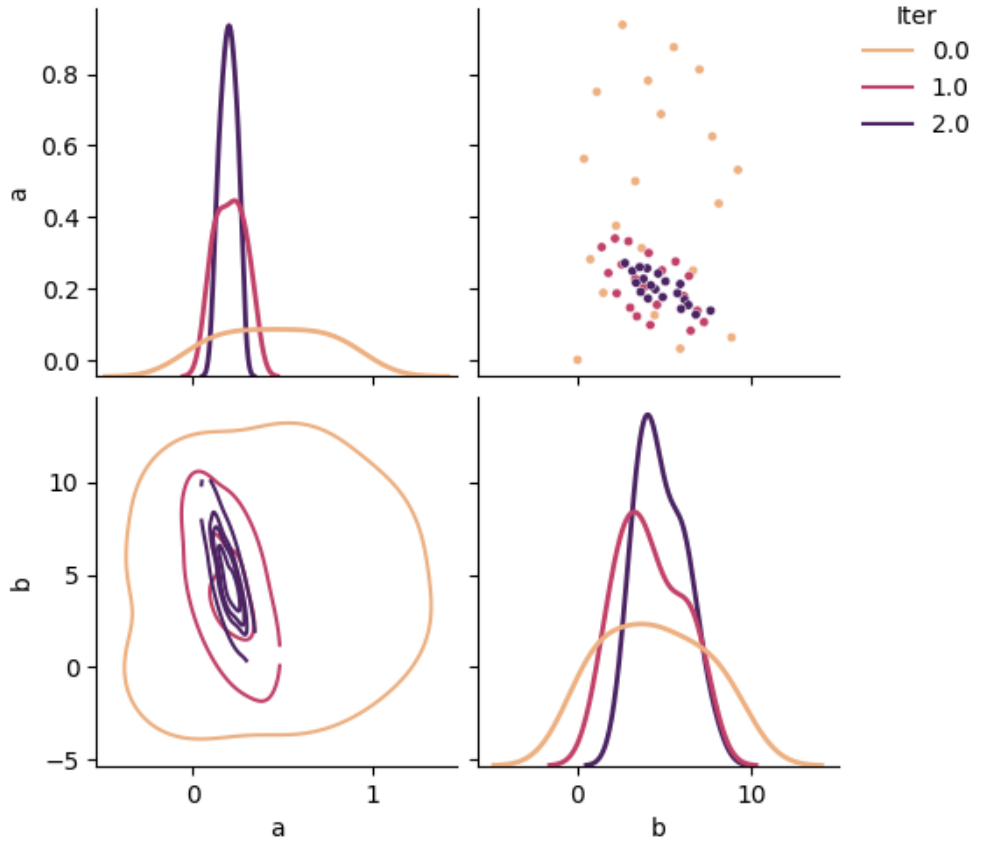


Fig. 11: Posterior distribution of the model parameters at various iterations.

Exercises

The examples in /examples/Bayesian_calibration/ show how to set up a Bayesian calibration for

- `particle-particle collision`
- `triaxial compression`

4.7.9 Particle-particle collision

First, create a synthetic reference dataset by running `the Yade script` with `yade Collision.py` and rename the output file `mv collision_test_run_sim.txt collision_obs.txt`. Then, run the script `python collision_calibration.py` to calibrate the DEM model parameters probabilistically. The entire calibration should take no more than 5 minutes.

1. Can you reduce the number of samples per iteration to 5 and still get a good result?
2. What about shifting the parameter bounds to [5, 0.0] and [9, 0.5]?

Since the simulation is fast, you can try different settings to see how they affect the calibration.

4.7.10 Triaxial compression

Run the script `yade triax_YADE_DEM_model.py` first to generate the reference data *triax_DEM_test_run_sim*. Then, run the script `python triax_calibration.py` to calibrate the DEM model parameters probabilistically. You should be able to get a decent result within two iterations of Bayesian calibration. On a 16-core machine, the calibration should take no more than 20 minutes.

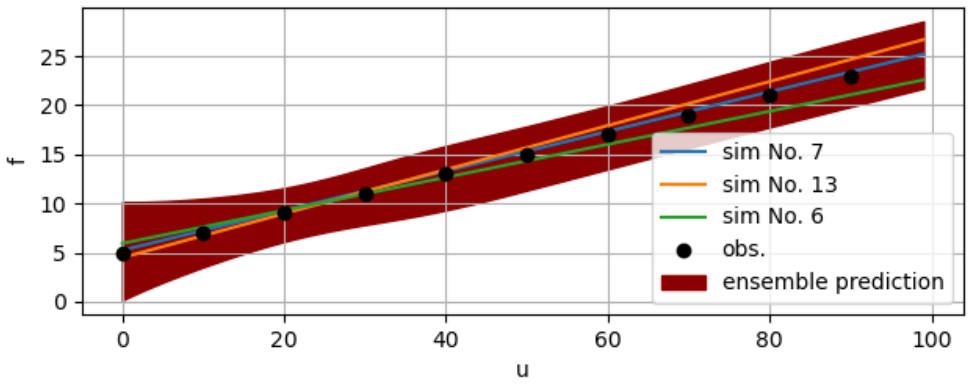


Fig. 12: Comparison between the reference and top three most probable simulation results.

1. Were you able to recover the original parameter values which you used to generate the reference data?
2. Can you decrease the number of samples per iteration to $3 * \mathrm{N} \log \mathrm{N}$ and still get a good agreement with the reference data?
3. GrainLearning uses variational Gaussian mixture model of scikit-learn to approximate the posterior distribution. Reduce the number of Gaussian components to 1 and see how that affects the calibration result.
4. Suppose you have done quite a few tests and would like to put them all together for GrainLearning to analyze. Have a look at the script `triax_calibration_load_and_run.py` to see how you could do that and even restart the calibration from where you left off.
5. For demonstration purposes, the DEM simulation does not include an initial step to check if the initial porosity is reached. Can you add this step to the DEM simulation and see how it affects the calibration?

These tutorials are extracted from the GrainLearning repository. More advanced tutorials can be found on https://grainlearning.readthedocs.io/.

Chapter 5

Performance enhancements

5.1 Accelerating Yade's FlowEngine with GPU

(Note: we thank Robert Caulk for preparing and sharing this guide)

5.1.1 Summary

This document contains instructions for adding Suite Sparse's GPU acceleration to Yade's Pore Finite Volume (PFV) scheme as demonstrated in [Caulk2019]. The guide is intended for intermediate to advanced Yade users. As such, the guide assumes the reader knows how to modify and compile Yade's source files. Readers will find that this guide introduces system requirements, installation of necessary prerequisites, and installation of the modified Yade. Lastly, the document shows the performance enhancement expected by acceleration of the factorization of various model sizes.

5.1.2 Hardware, Software, and Model Requirements

- **Hardware:**
  - CUDA-capable GPU with >3 GB memory recommended (64 mb required)
- **Software:**
  - NVIDIA CUDA Toolkit
  - SuiteSparse (CHOLMOD v2.0.0+)
  - Metis (comes with SuiteSparse)
  - CuBlas
  - OpenBlas
  - Lapack
- **Model:**
  - Fluid coupling (Pore Finite Volume aka Yade's "FlowEngine")
  - >10k particles, but likely >30k to see significant speedups
  - Frequent remeshing requirements

5.1.3 Install CUDA

The following instructions to install CUDA are a boiled down version of these instructions.

```
lspci | grep -i nvidia #Check your graphics card
# Install kernel headers and development packages
sudo apt-get install linux-headers-$(uname -r)
#Install repository meta-data (see **Note below):
sudo dpkg -i cuda-repo-<distro>_<version>_<architecture>.deb
sudo apt-get update  #update the Apt repository cache
sudo apt-get install cuda #install CUDA
# Add the CUDA library to your path
export PATH=/usr/local/cuda/bin${PATH:+:${PATH}}
export LD_LIBRARY_PATH=/usr/local/cuda/lib64\ ${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
```

Note: use this tool to determine your `<distro>_<version>_<architecture>` values.

Restart your computer.

Verify your CUDA installation by navigating to `/usr/local/cuda/samples` and executing the `make` command. Now you can navigate to `/usr/local/cuda/samples/1_Utilities/deviceQuery/` and execute `./deviceQuery` . Verify the `Result = PASS`.

5.1.4 Install OpenBlas, and Lapack

Execute the following command:

```
sudo apt-get install libopenblas-dev liblapack-dev
```

5.1.5 Install SuiteSparse

Download the SuiteSparse package and extract the files to `/usr/local/`. Run `make config` and verify `CUDART_LIB` and `CUBLAS_LIB` point to your cuda installed libraries. The typical paths will follow `CUDART_LIB=/usr/local/cuda-x.y/lib64` and `CUBLAS_LIB=/usr/local/cuda-x.y/lib64`. If the paths are blank, you may need to navigate to to `CUDA_PATH` in `/usr/local/SuiteSparse/SuiteSparse_config/SuiteSparse_config.mk` and modify it manually to point to your cuda installation. Navigate back to the main SuiteSparse folder and execute `make`. SuiteSparse is now compiled and installed on your machine.

Test CHOLMOD's GPU functionality by navigating to `SuiteSparse/CHOLMOD/Demo` and executing `sh gpu.sh`. Note: you will need to download the nd6k.mtx from here and put it in your home directory.

5.1.6 Compile Yade

Following the instructions outlined here, run `cmake` with `-DCHOLMOD_GPU=ON` and `-DSUITESPARSEPATH=/usr/local/SuiteSparse` (or your other custom path). Check the output to verify the paths to CHOLMOD (and dependencies such as AMD), SuiteSparse, CuBlas, and Metis are all identified as the paths we created when we installed these packages. Here is an example of the output you need to inspect:

```
-- Found Cholmod in /usr/local/SuiteSparse/lib/libcholmod.so
-- Found OpenBlas in /usr/lib/libopenblas.so
-- Found Metis in /usr/local/SuiteSparse/lib/libmetis.so
-- Found CuBlas in /usr/local/cuda-x.y/libcublas.so
-- Found Lapack in /usr/lib/liblapack.so
```

If you have multiple versions of any of these packages, it is possible the system finds the wrong one. In this case, you will need to either uninstall the old libraries (e.g. `sudo apt-get remove libcholmod` if the other library was installed with apt-get) or edit the paths within `cMake/Find_____.cmake`. If you installed a version of Cuda in a different location than `/usr/local`, you will need to edit `cMake/FindCublas.cmake` to reflect these changes before compilation.

Metis is compiled with SuiteSparse, so the Metis library and Metis include should link to files within `usr/local/SuiteSparse/`. When ready, complete installation with `make -jX install`. Keep in mind

that adding `CHOLMOD_GPU` alters `useSolver=4` so to work with the GPU and not the CPU. If you wish to `useSolver=4` with the CPU without unintended side effects (possible memory leaks), it is recommended to recompile with `CHOLMOD_GPU=OFF`. Of course, `useSolver=3` should always work on the CPU.

5.1.7 Controlling the GPU

The GPU accelerated solver can be activated within Yade by setting `flow.useSolver=4`. There are several environment variables that control the allowable memory, allowable GPU matrix size, etc. These are highlighted within the CHOLMOD User Guide, which can be found in `SuiteSparse/CHOLMOD/Doc`. At the minimum, the user needs to set the environment variable by executing `export CHOLMOD_USE_GPU=1`. It is also recommended that you designate half of your available GPU memory with `export CHOLMOD_GPU_MEM_BYTES=3000000000` (for a 6GB graphics card), if you wish to use the `multithread=True` functionality. If you have a multi-gpu setup, you can tell Yade to use one (or both GPUs with SuiteSparse-4.6.0-beta) by executing `export CUDA_VISIBLE_DEVICES=1`, where 1 is the GPU you wish to use.

5.1.8 Performance increase

[Catalano2012] demonstrated the performance of DEM+PFV coupling and highlighted its strengths and weaknesses. A significant strength of the DEM+PFV coupling is the asymptotic nature of triangulation costs, volume calculation costs, and force calculation costs ([Catalano2012], Figure 5.4). In other words, increasing the number of particles beyond ~200k results in negligible additional computational costs. The main weakness of the DEM+PFV coupling is the exponential increase of computational cost of factoring and solving increasingly larger systems of linear equations ([Catalano2012], Figure 5.7). As shown in Fig. *fig-cpuvsgpu*, the employment of Tesla K20 GPU decreases the time cost of factorization by up to 75% for 2.1 million DOFs and 356k particles.

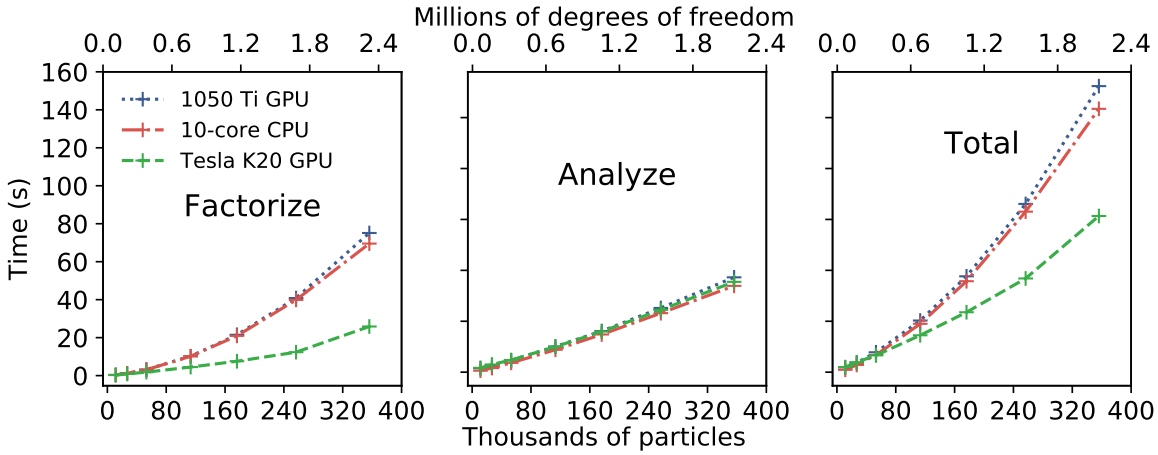


Fig. 1: Time required to factorize and analyze various sized matrices for 10-core CPU, 1050Ti GPU, and Tesla K20 GPU [Caulk2019].

Note: Tesla K20 5GB CPU + 10-core Xeon E5 2.8 GHz CPU

5.2 MPI parallelization

The module *mpy* implements parallelization by domain decomposition (distributed memory) using the Message Passing Interface (MPI) implemented by OpenMPI. It aims at exploiting large numbers of compute nodes by running independent instances of Yade on them. The shared memory and the distributed memory approaches are compatible, i.e. it is possible to run hybrid jobs using both, and it may well be the optimal solution in some cases.

Most (initially *all*) calls to OpenMPI library are done in Python using mpi4py. However for the sake of efficiency some critical communications are triggered via python wrappers of C++ functions, wherein messages are produced, sent/received, and processed.

This module development was started in 2018. It received contributions during a HPC hackathon. An extension enables *parallel coupling with OpenFoam*.

Note

see also *reference documentation of the mpy* module.

Note

Disclaimer: even though the *yade.mpy* module provides the function *mpirun*, which may seem as a simple replacement for *O.run()*, setting up a simulation with mpy might be deceptively triavial. As of now, it is anticipated that, in general, a simple replacement of "run" by "mpirun" in an arbitrary script will not speedup anything and may even fail miserably (it could be improved in the future). To understand why, and to tackle the problems, basic knowledge of how MPI works will certainly help (specifically mpi4py).

5.2.1 Concepts

subdomain: a (sub)set of bodies attached to one MPI process after domain decomposition - with or without spatial coherence. The corresponding class in Yade is *Subdomain*, a *Shape* instance with helper functions for MPI communications. In some sense *Subdomain* is to subscribed bodies what *Clump* (another *Shape*) is to clump members.

rank: subdomain index from 0 to N-1 (with N the number of mpi processes) to identify subdomains. The rank of the subdomain a body belongs to can be retrieved as *Body.subdomain*. Each subdomain corresponds to an instance of Yade and a specific scene during parallel execution. The rank of the scene is given by *Scene.subdomain*.

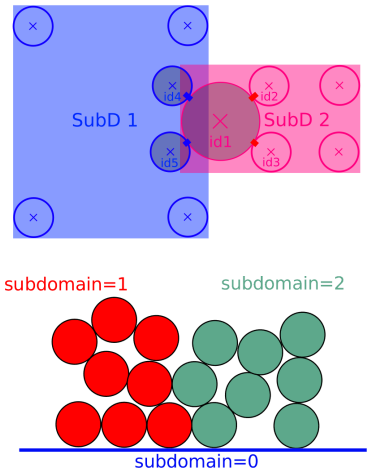
master: refers to subdomain with *rank* =0. This subdomain does not behave like others. In general master will handle boundary conditions and it will control transitions and termination of the whole simulation. Unlike standard subdomains it may not contain a large number of raw bodies (i.e. not beyond objects bounding the scene such as walls or boxes). In interactive execution master is the process responding to the python prompt.

splitting and merging: cutting a master *Scene* into a set of smaller, distributed, scenes is called "splitting". The split is undone by a 'merge', by which all bodies and (optionally) all interactions are sent back to the master thread. Splitting, running, then merging, should leave the scene as if no MPI had been used at all (i.e. as if the same number of iterations had been executed in single-thread). Therefore normal O.run() after that should work as usual.

intersections: subsets of bodies in a subdomain intersected by the bounding box of other subdomains (see *fig-subdomains*). *intersection(i,j)* refers to the bodies owned by current (i) subdomain and intersecting subdomain j (retrieved as *O._sceneObj.subD.intersections[j]*); *mirrorIntersection(i,j)* refers to bodies owned by j and intersecting current domain (retrieved as *O._sceneObj.subD.mirrorIntersections[j]*). The bodies are listed by *Body.id*. By definition *intersection(i,j)=mirrorIntersection(j,i)*.

The intersections and mirror intersections are updated automatically as part of parallel collision detection. They define which body states need to be communicated. The bodies in intersections need to be *sent* to other subdomains (in pratice only updated position and velocity are sent at every iteration), the bodies in mirrorIntersections need to be received from other subdomains.

Two overlapping subdomains and their intersections. In this situation we have *SubD1.intersections[SubD2.subdomain]=[id4,id5]* and *SubD1.mirrorIntersections[SubD2.subdomain]=[id1]*, with *SubD1* and *SubD2* instances of *Subdomain*.



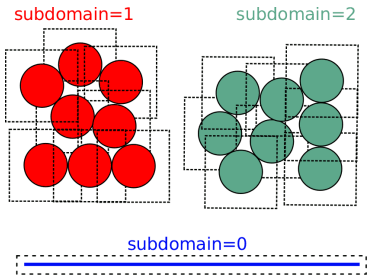
5.2.2 Walkthrough

For demonstrating the main internal steps in the implemented parallel algorithm let us consider the example script examples/mpi/testMPI_2D.py. Executing this script (interactive or passive mode) with three MPI processes generates the scene as shown in *fig-scene-mpi*. It then executes *mpirun*, which triggers the steps described hereafter.

In this scene, we have three MPI processes (three subdomains) and the raw bodies are partitioned among the subdomains/ranks 1 and 2. The master process with subdomain=0 holds the boundary/wall type body. Bodies can be manually assigned or automatically assigned via a domain decomposition algorithm. Details on the dommain decomposition algorithm is presented in the later section of this document.

Scene splitting :

In the function *mpy.splitScene*, called at the beginning of mpi execution, specific engines are added silently to the scene in order to handle what will happen next. That very intrusive operation can even change settings of some pre-existing engines, in particular *InsertionSortCollider*, to make them behave with MPI-friendlyness. *InsertionSortCollider.verletDist* is an important factor controlling the efficiency of the simulations. The reason for this will become evident in the later steps.

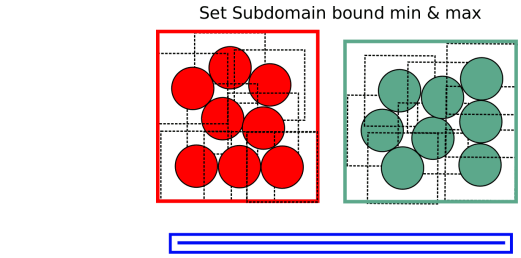


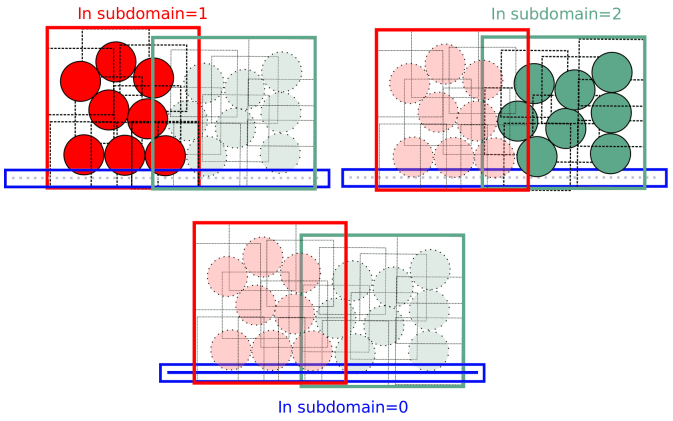
Bounds dispatching : In the next step, the *Body.bound* is dispatched with the *Aabb* extended as shown in figure *fig-regularbounds* (in dotted lines). Note that the *Subdomain Aabb* is obtained from taking the min and max of the owned bodies, see figure *fig-subDBounds* with solid coloured lines for the subdomain *Aabb*. At this time, the min and max of other subdomains are unknown.

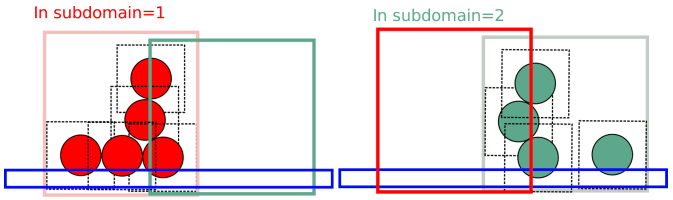
Update of Domain bounds : Once the bounds for the regular bodies and the *local subdomain* has been dispatched, information on the other subdomain bounds are obtained via the function *mpy.updateDomainBounds*. In this collective communication, each subdomain broadcasts its *Aabb.min* and *Aabb.max* to other subdomains. Figure *fig-subdomain-bounds* shows a schematic in which each subdomain has received the *Aabb.min* and *Aabb.max* of the other subdomains.

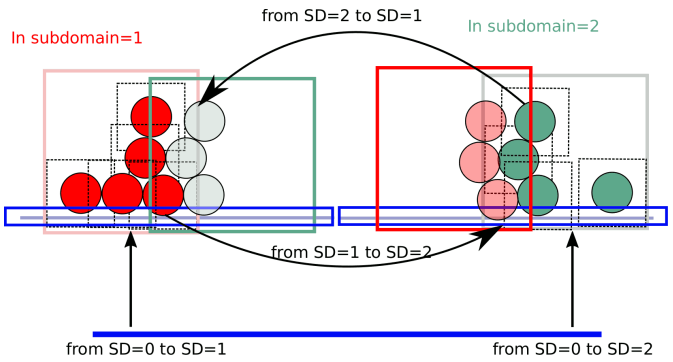
Parallel Collision detection :

- Once the *Aabb.min* and *Aabb.max* of the other subdomains are obtained, the collision detection algorithm is used to determine the bodies that have intersections with the remote subdomains. The ids of the identified bodies are then used to build the *Subdomain.intersections* list.

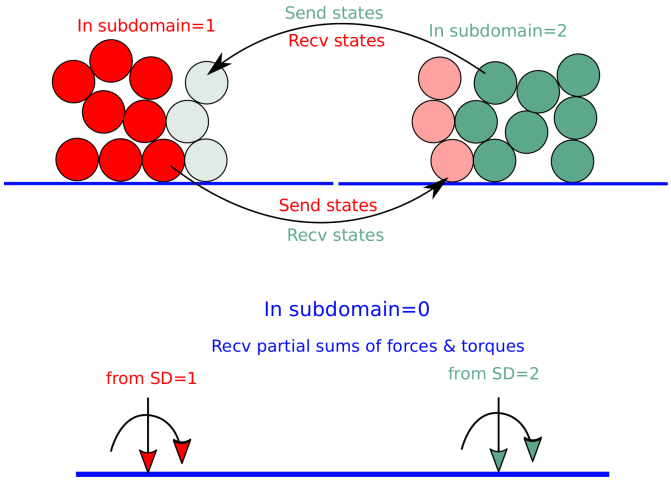

Set Subdomain bound min & max


In subdomain=1
In subdomain=2
In subdomain=0


In subdomain=1
In subdomain=2


from SD=2 to SD=1
In subdomain=1
In subdomain=2
from SD=1 to SD=2
from SD=0 to SD=1
from SD=0 to SD=2

- Next step involves obtaining the ids of the remote bodies intersecting with the current subdomain (*Subdomain.mirrorIntersections*). Each subdomain sends its list of local body intersections to the respective remote subdomains and also receives the list of intersecting ids from the other subdomains. If the remote bodies do not exist within the current subdomain's *BodyContainer*, the subdomain then *requests* these remote bodies from the respective subdomain. A schematic of this operation is shown in figure *fig-mirrorIntersections*, in which subdomain=1 receives three bodies from subdomain=2, and 1 body from subdomain=0. subdomain=2 receives three bodies from subdomain=1. subdomain=0 only sends its bodies and does *not* receive from the worker subdomains. This operation sets the stage for communication of the body states to/from the other subdomains.



Update states :

Once the subdomains and the associated intersecting bodies, and remote bodies are identified, *State* of these bodies are sent and received every timestep, by peer-to-peer communications between the interacting subdomains. In the case of an interaction with the master subdomain (subdomain=0), only the total force and torque exerted on master's bodies by a given subdomain are sent. Figure *fig-sendRecvStates* shows a schematic in which the states of the remote bodies between subdomain=1 and subdomain=2 are communicated. Subdomain=0 receives forces and torques from subdomain=1 and subdomain=2.

5.2.3 MPI initialization and communications

The mpy modules tries to retain one of Yade's most important features: interactive access to the objects of scene (or of multiple scenes in this case), as explained below. Interactive execution does not use the *mpiexec* command of OpenMPI, instead, a pool of workers is spawned by the mpy module after Yade startup. In production one may use passive jobs, and in that case *mpiexec* will preceed the call to Yade.

Note

Most examples in this page use 4 mpi processes. It is not a problem, in principle, to run the examples even if the number of available cores is less than 4 (this is called oversubscribing (it may also fail depending on OS and MPI implementation). There is no performance gain to expect from oversubscribing but it is useful for experiments (e.g. for testing the examples in this page on a single-core machine).

Interactive mode

The interactive mode aims primarily at inspecting the simulation after some MPI execution for debugging. Functions shown here (especially *sendCommand*) may also be usefull in the general case, to achieve advanced tasks such as controlling transitions between phases of a simulation, collecting and processing results.

Explicit initialization from python prompt

A pool of Yade instances can be spawned with mpy.initialize() as illustrated hereafter. Mind that the next sequences of commands are supposed to be typed directly in the python prompt after starting Yade, it will not give exactly the same result if it is pasted into a script executed by Yade (see the next section on automatic initialization):

```
@suppress
Yade [1]: from yade.utils import *

@suppress
Yade [1]: O.engines=yade.utils.defaultEngines

Yade [2]: wallId=O.bodies.append(box(center=(0,0,0),extents=(2,0,1),fixed=True))

Yade [3]: for x in range(-1,2):
   ...:    O.bodies.append(sphere((x,0.5,0),0.5))
   ...:

Yade [5]: from yade import mpy as mp

@suppress
Yade [5]: mp.COLOR_OUTPUT=False

@doctest
Yade [6]: mp.initialize(4)
Master: I will spawn  3  workers
->  [6]: (0, 4)
```

After mp.initialize(np) the parent instance of Yade takes the role of master process (rank=0). It is the only one executing the commands typed directly in the prompt. The other instances (rank=1 to rank=np-1) are idle and they wait for commands sent from master. Sending commands to the other instances can be done with *mpy.sendCommand()*, which by default returns the result or the list of results. We use that command below to verify that the spawned workers point to different (still empty) scenes:

```
Yade [8]: len(O.bodies)
 ->  [8]: 4

Yade [10]: mp.sendCommand(executors="all",command="len(O.bodies)",wait=True) #check␣
↪content
->  [10]: [4, 0, 0, 0]

Yade [9]: mp.sendCommand(executors="all",command="str(O)") # check scene pointers
 ->  [9]:
['<yade.wrapper.Omega object at 0x7f9c0a399490>',
'<yade.wrapper.Omega object at 0x7f9231213490>',
'<yade.wrapper.Omega object at 0x7f20086a1490>',
'<yade.wrapper.Omega object at 0x7f622b47f490>']
```

Sending commands makes it possible to manage all types of message passing using calls to the underlying mpi4py (see mpi4py documentation). Be carefull with sendCommand "blocking" behavior by default. Next example would hang without "wait=False" since both master and worker would be waiting for a message from each other.

```
Yade [3]: mp.sendCommand(executors=1,command="message=comm.recv(source=0); print(
↪'received',message)",wait=False)

Yade [4]: mp.comm.send("hello",dest=1)
```

(continues on next page)

(continued from previous page)

```
received hello
```

Every picklable python object (namely, nearly all Yade objects) can be transmitted this way. Remark hereafter the use of *mpy.mprint* (identifies the worker by number and by font colors). Note also that the commands passed via *sendCommand* are executed in the context of the mpy module, for this reason *comm*, *mprint*, *rank* and all objects of the module are accessed without the *mp.* prefix.

```
Yade [3]: mp.sendCommand(executors=1,command="O.bodies.append(comm.recv(source=0))",
 ↪wait=False) # leaves the worker idle waiting for an argument to append()

Yade [4]: b=Body(shape=Sphere(radius=0.7))  # now create body in the context of master

Yade [5]: mp.comm.send(b,dest=1) # send it to worker 1

Yade [6]: mp.sendCommand(executors="all",command="mprint('received',[b.shape.radius␣
 ↪if hasattr(b.shape,'radius') else None for b in O.bodies])")
Master: received [None, 0.5, 0.5, 0.5]
Worker1: received [0.7]
Worker3: received []
Worker2: received []
->  [5]: [None, None, None, None] # printing yields no return value, hence that empty␣
 ↪list of returns, "wait=False" argument to sendCommand would suppress it
```

Explicit initialization from python script

Though usefull for advanced operations, the function sendCommand() is limited. Basic features of the python language are missing, e.g. function definitions and loops are a problem - in fact every code fragment which can't fit on a single line is. In practice the mpy module provides a mechanism to initialize from a script, where functions and variables will be declared.

Whenever Yade is started with a script as an argument, the script name will be remembered, and if mpy.initialize() is called (by the script itself or interactively in the prompt), all Yade instances will be initialized with that same script. It makes distributing function definitions and simulation parameters trivial (and even distributing scene constructions as seen below).

This behaviour is what happens usually with MPI: all processes execute the same program. It is also what happens with "mpiexec -np N yade …".

If the first commands above are pasted into a script used to start Yade, all workers insert the same bodies as master (with interactive execution only master was inserting). Here is the script:

```
# script 'test1.py'
wallId=O.bodies.append(box(center=(0,0,0),extents=(2,0,1),fixed=True))
for x in range(-1,2):
        O.bodies.append(sphere((x,0.5,0),0.5))
from yade import mpy as mp
mp.initialize(4)
print( mp.sendCommand(executors="all",command="str(O)",wait=True) )
print( mp.sendCommand(executors="all",command="len(O.bodies)",wait=True) )
```

and the output reads:

```
yade test1.py
...
Running script test1.py
Master: will spawn  3  workers
None
None
```

(continues on next page)

(continued from previous page)

```
None
None
None
None
['<yade.wrapper.Omega object at 0x7feb979403a0>', '<yade.wrapper.Omega object at␣
↪0x7f5b61ae9440>', '<yade.wrapper.Omega object at 0x7fdd466b8440>', '
↪<yade.wrapper.Omega object at 0x7f8dc7b73440>']
[4, 4, 4, 4]
```

That's because all instances execute the script in the initialize() phase. "None" is printed 2x3 times because the script contains *print(mp.sendCommand(...))* twice, the workers try to execute that too, but for them *sendCommand* returns by default, hence the None.

Though logical, this result is not what we want if we try to split a simulation into pieces. The solution (typical of all mpi programs) is to use the *rank* of the process in conditionals. Different parts of the script can then be executed, differently, by each worker, depending on its rank. In order to produce the same result as before, for instance, the script can be modified as follows:

```
# script 'test2.py'
from yade import mpy as mp
mp.initialize(4)
if mp.rank==0: # only master
        wallId=O.bodies.append(box(center=(0,0,0),extents=(2,0,1),fixed=True))
        for x in range(-1,2):
        O.bodies.append(sphere((x,0.5,0),0.5))

        print( mp.sendCommand(executors="all",command="str(O)",wait=True) )
        print( mp.sendCommand(executors="all",command="len(O.bodies)",wait=True) )
        print( mp.sendCommand(executors="all",command="str(O)",wait=True) )
```

Resulting in:

```
Running script test2.py
Master: will spawn  3  workers
['<yade.wrapper.Omega object at 0x7f21a8c8d3a0>', '<yade.wrapper.Omega object at␣
↪0x7f3142e43440>', '<yade.wrapper.Omega object at 0x7fb699b1a440>', '
↪<yade.wrapper.Omega object at 0x7f1e4231e440>']
[4, 0, 0, 0]
```

We could also use *rank* to assign bodies from different regions of space to different workers, as found in example examples/mpi/helloMPI.py, with rank-dependent positions:

```
# rank is accessed without "mp." prefix as it is interpreted in mpy module's scope
mp.sendCommand(executors=[1,2],command=
↪"ids=O.bodies.append([sphere((xx,1.5+rank,0),0.5) for xx in range(-1,2)])")
```

Keep in mind that the position of the call *mp.initialize(N)* relative to the other commands has no consequence for the execution by the workers (for them initialize() just returns), hence program logic should not rely on it. The workers execute the script from begin to end with the same MPI context, already set when the first line is executed. It can lead to counter intuitive behavior, here is a script:

```
# testInit.py
# script.py
O.bodies.append([Body() for i in range(100)])

from yade import mpy as mp
mp.mprint("before initialize: rank ", mp.rank,"/", mp.numThreads,"; ",len(O.bodies),"␣
```

(continues on next page)

(continued from previous page)

```
↪bodies")
mp.initialize(2)
mp.mprint("after initialize: rank ", mp.rank,"/", mp.numThreads,"; ",len(O.bodies),"␣
↪bodies")
```

and the output:

```
Running script testInit.py
Master: before initialize: rank  0 / 1 ;  100  bodies
Master: will spawn  1  workers
Master: after initialize: rank  0 / 2 ;  100  bodies
Worker1: before initialize: rank  1 / 2 ;  100  bodies
Worker1: after initialize: rank  1 / 2 ;  100  bodies
```

mpirun (automatic initialization)

Effectively running a distributed DEM simulation on the basis of the previously described commands would be tedious. The mpy module thus provides the function *mpy.mpirun* to automate most of the steps, as described in *introduction*. Mainly, splitting the scene into subdomains based on rank assigned to bodies and handling collisions between the subdomains as time integration proceeds (includes changing the engine list agressively to make this all happen).

If needed, the first execution of mpirun will call the function initialize(), which can therefore be omitted on the user's side. The subdomains will be merged into a centralized scene on the master process at the end of the iterations depending on the argument *withMerge*.

Here is a concrete example where a floor is assigned to master and multiple groups of spheres are assigned to subdomains:

```
import os
from yade import mpy as mp

NSTEPS=5000 #turn it >0 to see time iterations, else only initilization
numThreads = 4 # number of threads to be spawned, (in interactive mode).

#materials
young = 5e6
compFricDegree = 0.0
O.materials.append(FrictMat(young=young, poisson=0.5, frictionAngle =␣
↪radians(compFricDegree), density= 2600, label='sphereMat'))
O.materials.append(FrictMat(young=young*100, poisson = 0.5, frictionAngle =␣
↪compFricDegree, density =2600, label='wallMat'))


#add spheres

mn,mx=Vector3(0,0,0),Vector3(90,180,90)
pred = pack.inAlignedBox(mn,mx)
O.bodies.append(pack.regularHexa(pred,radius=2.80,gap=0, material='sphereMat'))

#walls (floor)

wallIds=aabbWalls([Vector3(-360,-1,-360),Vector3(360,360,360)],thickness=10.0,␣
↪material='wallMat')
O.bodies.append(wallIds)

#engines
```

(continues on next page)

(continued from previous page)

```
O.engines=[
        ForceResetter(),
        InsertionSortCollider([
                Bo1_Sphere_Aabb(),
                Bo1_Box_Aabb()], label = 'collider'), # always add labels.
        InteractionLoop(
                [Ig2_Sphere_Sphere_ScGeom(),Ig2_Box_Sphere_ScGeom()],
                [Ip2_FrictMat_FrictMat_FrictPhys()],
                [Law2_ScGeom_FrictPhys_CundallStrack()],
                label="interactionLoop"
        ),
        GlobalStiffnessTimeStepper(timestepSafetyCoefficient=0.3, ␣
↪timeStepUpdateInterval=100, parallelMode=True, label = 'timeStepper'),
        NewtonIntegrator(damping=0.1,gravity = (0, -0.1, 0), label='newton'),
        VTKRecorder(fileName='spheres/3d-vtk-', recorders=['spheres', 'intr', 'boxes
↪'], parallelMode=True,iterPeriod=500), #use .pvtu to open spheres, .pvtp for ints,␣
↪and .vtu for boxes.
]

#set a custom verletDist for efficiency.
collider.verletDist = 1.5

#########  RUN  ##########
# customize mpy
mp.ERASE_REMOTE_MASTER = True   #keep remote bodies in master?
mp.DOMAIN_DECOMPOSITION= True   #automatic splitting/domain decomposition
#mp.mpirun(NSTEPS)              #passive mode run
mp.MERGE_W_INTERACTIONS = False
mp.mpirun(NSTEPS,numThreads,withMerge=True) # interactive run, numThreads is the␣
↪number of workers to be initialized, see below for withMerge explanation.
mp.mergeScene()  #merge scene after run.
if mp.rank == 0: O.save('mergedScene.yade')

#demonstrate getting stuff from workers, here we get kinetic energy from worker␣
↪subdomains, notice that the master (mp.rank = 0), uses the sendCommand to tell␣
↪workers to compute kineticEnergy.
if mp.rank==0:
        print("kinetic energy from workers: "+str(mp.sendCommand([1,2],
↪"kineticEnergy()",True)))
```

The script is then executed:

```
yade script.py
```

For running further timesteps, the mp.mpirun command has to be executed in yade prompt:

```
Yade [0]: mp.mpirun(100,4,withMerge=False) #run for 100 steps and no scene merge.

Yade [1]: mp.sendCommand([1,2],"kineticEnergy()",True) # get kineticEnergy from␣
↪workers 1 and 2.

Yade [2]: mp.mpirun(1,4,withMerge=True) # run for 1 step and merge scene into master.␣
↪Repeat multiple time to watch evolution in QGL view
```

Non-interactive execution

Instead of spawning mpi processes after starting Yade, it is possible to run Yade with the classical "mpiexec" from OpenMPI. Importantly, it may be the only method allowed through HPC job submission systems. When using mpiexec there is no interactive shell, or a broken one (which is ok in general in production). The job needs to run (or "*mpirun*") and terminate by itself.

The functions *initialize* and *mpirun* described above handle both interactive and passive executions transparently, and the user scripts should behave the same in both cases. "Should", since what happens behind the scenes is not exactly the same at startup, and it may surface in some occasions (let us know).

Provided that a script calls *mpy.mpirun* with a number of timesteps, the simulation (see e.g. examples/mpi/vtkRecorderExample.py) is executed with the following command:

```
mpiexec -np NUMSUBD+1 yade vtkRecorderExample.py
```

where *NUMSUBD* corresponds to the required number of subdomains.

Note

Remember that the master process counts one while it does not handle an ordinary subdomain, therefore the number of processes is always *NUMSUBD* +1.

5.2.4 Splitting

Splitting an initial scene into subdomains and updating the subdomains after particle motion are two critical issues in terms of efficiency. The decomposition can be prescribed on users's side (first section below), but mpy module also provides algorithms for both.

Note

The mpy module has no requirement in terms of how the subdomains are defined, and using the helper functions described here is not a requirement. Even assigning the bodies randomly from a large cloud to a number of subdomains (such that the subdomains overlap each other and the scene entirely) would work. It would only be suboptimal as the number of interactions between subdomains would increase compared to a proper partition of space.

Split by yourself

In order to impose a decomposition it is enough to assign *Body.subdomain* a value corresponding to the process rank it should belong to. This can be done either in one centralized scene that is later split, or by inserting the correct subsets of bodies independently in each subdomain (see section on scene construction)

In the example script examples/mpi/testMPI_2D.py the spheres are generated as follows (centralized construction in this example, easily turned into distributed one). For each available worker a bloc of spheres is generated with a different position in space. The spheres in each block are assigned a subdomain rank (and a color for visualisation) so that they will be picked up by the right worker after mpirun().:

```
for sd in range(0,numThreads-1):
        col = next(colorScale)
        ids=[]
        for i in range(N):#(numThreads-1) x N x M spheres, one thread is for master␣
↪and will keep only the wall, others handle spheres
                for j in range(M):
                        id = O.bodies.append(sphere((sd*N+i+j/30.,j,0),0.500,
↪color=col)) #a small shift in x-positions of the rows to break symmetry
```

(continues on next page)

(continued from previous page)

```
                    ids.append(id)
                for id in ids: O.bodies[id].subdomain = sd+1
```

Don't know how to split? Leave it to mpirun

Initial split

mpirun will decide by itself how to distribute the bodies across several subdomains if *DOMAIN_DECOMPOSITION* =True. In such case the difference between the sequential script and its mpi version is limited to importing mpy and calling mpirun after turning the *DOMAIN_DECOMPOSITION* flag.

The automatic splitting of bodies to subdomains is based on the Orthogonal Recursive Bisection Algortithm of Berger [Berger1987], and [Fleissner2007]. The partitioning is based on bisecting the space at several *levels*, with the longest axis in each level chosen as the bisection axis. The number of levels is determined as $\mathtt{int}(\log_2(N_w))$ with N_w being the number of worker subdomains. A schematic of this decomposition is shown in *fig-bisectionAlgo*, with 4 worker subdomains. At the initial stage (level = 0), we assume that subdomain=1 contains the information of the body positions (and bodies), the longest axis is first determined, this forms the bisectioning axis/plane. The list containing the body positions is sorted along the bisection axis, and the median of this sorted list is determined. The bodies with positions (bisection coordinate) less than the median is coloured with the current subdomain, (SD=1) and the other half is coloured with SD = 2, the subdomain colouring at each level is determined using the following rule:

```
if (subdomain <  1<<level) : this subdomain gets the bodies with position␣
↪lower than the median.
if ((subdomain >  1<<level) and (subdomain <  1<<(level+1) ) ) : this␣
↪subdomain gets the bodies with position greater than median, from␣
↪subdomain - (1<<level)
```

This process is continued until the number of levels are reached.

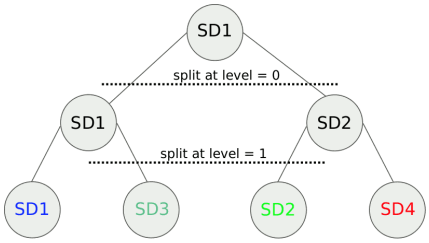
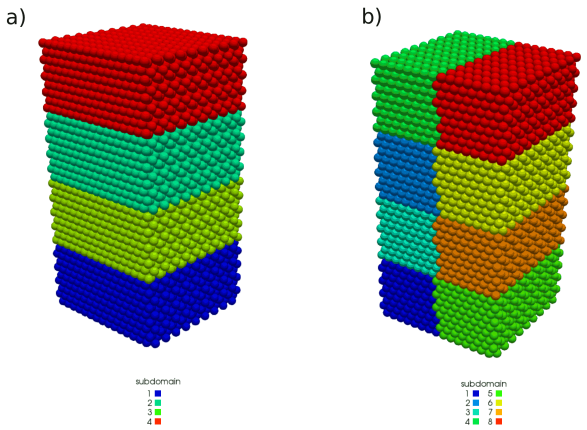


Figure *fig-domainDecompose* shows the resulting partitioning obtained using the ORB algorithm : (a) for 4 subdomains, (b) for 8 subdomains. Odd number of worker subdomains are also supported with the present implementation.

The present implementation can be found in py/bisectionDecomposition.py, and a parallel version can be found here.



Note

importing py/bisectionDecomposition.py triggers the import of *mpi4py* and ultimately of the MPI library and related environment variables. The *mpy* module may change some mpi settings on import, therefore it is safer to only import *bisectionDecomposition* after some `mpy.initialize()`.

Updating the decomposition (load balancing)

As the bodies move, each subdomain may experience overall distortion and diffusion of bodies leading to an undesirable overlap between multiple subdomains. This subdomain overlap introduces inefficiencies in communications between MPI workers, and thus we aim to keep the subdomains as compact as possible by using an algorithm that dynamically reallocates bodies to new subdomains with an objective of minimizing MPI communications. The algorithm exploits *InsertionSortCollider* to reassign bodies efficiently and in synchronicity with collision detection, and it can be activated if *mpy.REALLOCATE_FREQUENCY* >0.

The algorithm is *not* centralized, which preserves scalability. Additionally, it only engages peer-to-peer communications between MPI workers that share an intersection. The re-assignment depends on a filter for making local decisions. At the moment, there is one filter available called *mpy.medianFilter*. Custom filters can be used instead.

The median filter body re-allocation criterion criterion involves finding the position of a median plane between two subdomains such that after discriminating bodies on the "+" and "-" side of that plane the total number in each subdomain is preserved. It results in the type of split shown in the video hereafter. Even though the median planes seem to rotate rather quickly at some point in this video, there are actually five collision detections between each re-allocation, i.e. thousands of time iterations to effectively rotate the split between two different colors. These progressive rotations are beneficial since the initial split would have resulted in flat discs otherwise.

Note

This is not a load balancing in the sense of achieving an equal amount of work per core. In fact that sort of balancing is achieved by definition already as soon as each worker is assigned the same amount of bodies (and because a subdomain is really ultimately a list of bodies, not a specific region of space). Instead the objective is to decrease the communication times overall.

Centralized versus distributed scene construction

For the centralized scene construction method, the master process creates all of the bodies of a scene and assigns subdomains to them. As part of mpy initialization some engines will be modified or inserted, then the scene is broadcasted to the workers. Each worker receives the entire scene, identifies its assigned bodies via *Body.subdomain* (if worker's `rank==b.subdomain` the bodies are retained) and erase the others. Such a scene construction was used in the previous example and it is by far the simplest. It makes no real difference with building a scene for non-MPI execution besides calling *mp.mpirun* instead or just *O.run*.

For large number of bodies and processes, though, the centralized scene construction and distribution can consume a significant amount of time. It can also be memory bound since the memory usage is quadratic: suppose N bodies per thread on a 32-core node, centralized construction implies that 32

user is in charge of setting up the subdomains and partitioning the bodies. An example of distributed insertion can be found in examples/mpi/parallelBodyInsert3D.py.

The relevant fragment, where the filtering is done by skipping all steps of a loop except the one with proper rank (keep in mind that all workers will run the same loop but they all have a different rank each), reads:

```
#add spheres
subdNo=0
import itertools
_id = 0 #will be used to count total number of bodies regardless of subdomain␣
↪attribute, so that same ids are not reused for different bodies
for x,y,z in itertools.product(range(int(Nx)),range(int(Ny)),range(int(Nz))):
        subdNo+=1
        if mp.rank!=subdNo: continue
        ids=[]
        for i in range(L):#(numThreads-1) x N x M x L spheres, one thread is for␣
↪master and will keep only the wall, others handle spheres
                for j in range(M):
                        for k in range(N):
                                dxOndy = 1/5.; dzOndy=1/15.  # shifts in␣
↪x/y-positions to make columns inclines
                                px= x*L+i+j*dxOndy; pz= z*N+k+j*dzOndy; py =␣
↪(y*M+j)*(1 -dxOndy**2 -dzOndy**2)**0.5 #so they are always nearly touching initialy
                                id = O.bodies.
↪insertAtId(sphere((px,py,pz),0.500),_id+(N*M*L*(subdNo-1)))
                                _id+=1
                                ids.append(id)
        for id in ids: O.bodies[id].subdomain = subdNo

if mp.rank==0: #the wall belongs to master
        WALL_ID=O.bodies.insertAtId(box(center=(Nx*L/2,-0.5,Nz*N/2),extents=(2*Nx*L,0,
↪2*Nz*N),fixed=True),(N*M*L*(numThreads-1)))
```

The bissection algorithm can be used for defining the initial split, in the distributed case too, since it takes a points dataset as input. Provided that all workers work with the same dataset (e.g. the same sequence of a random number generator) they will all reach the same partitioning, and they can instantiate their bodies on this basis.

5.2.5 Merging

The possibility of a "merge", shown in the previous example, can be performed using an optional argument of *mpirun* or as a standalone function *mpy.mergeScene* .

If withMerge=True in mpirun then the bodies in master scene are updated to reflect the evolution of their distributed clones. This is done once after finishing the required number of iterations in mpirun. This *merge* operation can include updating interactions. *mpy.mergeScene* does the same within the current iteration. Merging is an expensive task which requires the communication of large messages and, therefore, it should be done purposely and at a reasonable frequency. It can even be the main bottleneck for massively parallel scenes. Nevertheless, it can be useful for debugging with the 3D view, or for various post-processing tasks. The *MERGE_W_INTERACTIONS* provides a full merge, i.e. the interactions in the worker subdomains and between the subdomains are included, otherwise, only the position and states of the bodies are used. Merging with interactions should result in a usual Yade scene, ready for further time-stepping in non-mpi mode or (more useful) for some post-processing. The merge operation is not required for a proper time integration in general.

5.2.6 Hints and problems to expect

MPI support in engines

For MPI cases, the *parallelMode* flag for *GlobalStiffnessTimeStepper* and *VTKRecorder* have to be turned on. They are the only two engines upgraded with MPI support at the moment.

For other things. Read next section and be careful. If you feel like implementing MPI support for other engines, that would be great, consider using the two available examples as guides. Let us know!

Reduction (partial sums)

Quantities such as kinetic energy cannot be obtained for the entire scene just by summing the return value of *kineticEnergy()* from each subdomain. This is because each subdmomain may contain also images of bodies from intersecting subdomains and they may add their velocity, mass, or whatever is summed, to what is returned by each worker. Although some most-used functions of Yade may progressively get mpi support to filter out bodies from remote domains, it is not standard yet and therefore partial sums may need to be implemented on a case-by-case basis, with proper filtering in the user script.

This is just an example of why many things may go wrong if *run* is directly replaced by *mpirun* in a complex script.

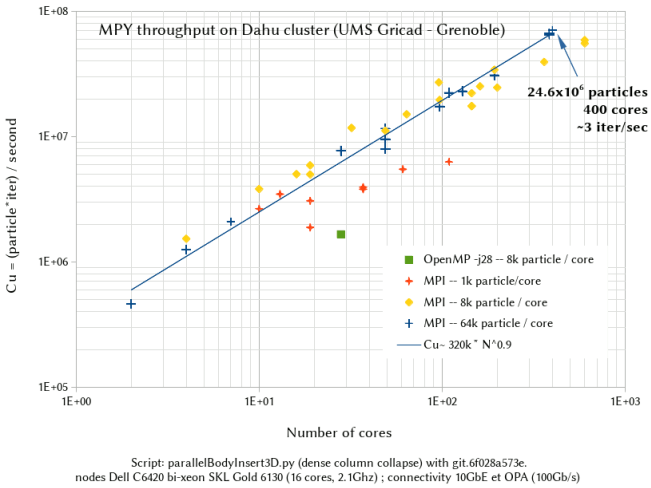
Miscellaneous

- sendCommand() has a hardcoded latency of 0.001s to not keep all cores 100% busy waiting for a command (with possibly little left to OS). If sendCommand() is used at high frequency in complex algorithms it might be beneficial to decrease that sleep time.

5.2.7 Control variables

- VERBOSE_OUTPUT : Details on each *operation/step* (such as *mpy.splitScene*, *mpy.parallelCollide* etc) is printed on the console, useful for debugging purposes
- ACCUMULATE_FORCES : Control force summation on bodies owned by the master.
- ERASE_REMOTE_MASTER : Erase remote bodies in the master subdomain or keep them as unbounded ? Useful for fast merge.
- OPTIMIZE_COM, USE_CPP_MPI : Use optimized communication functions and MPI functions from *Subdomain* class
- YADE_TIMING : Report timing statistics, prints time spent in communications, collision detection and other operations.
- DISTRIBUTED_INSERT : Bodies are created and inserted by each subdomain, used for distributed scene construction.
- DOMAIN_DECOMPOSITION : If true, the bisection decomposition algorithm is used to assign bodies to the workers/subdomains.
- MINIMAL_INTERSECTIONS : Reduces the size of position/velocity communications (at the end of the colliding phase, we can exclude those bodies with no interactions besides body<->subdomain from intersections).
- REALLOCATE_FREQUENCY : if > 0, bodies are migrated between subdomains for efficient load balancing. If =1 realloc. happens each time collider is triggered, else every N collision detection
- REALLOCATE_MINIMAL : Intersections are minimized before reallocations, hence minimizing the number of reallocated bodies
- USE_CPP_REALLOC : Use optimized C++ functions to perform body reallocations
- FLUID_COUPLING : Flag for coupling with OpenFOAM.

5.2.8 Benchmark



Comments:

- From 1k particles/core to 8k particles/core there is a clear improvement. Obviously 1k is too small and most of the time is spent in comunications.
- From 8k/core to 64k/core the throughput per core is more or less the same, and the performance is not too far from linear. The data includes elimination of random noise, and overall it is not clear to me which non-linearity comes from the code and which one comes from the hardware.
- Conclusion, if you don't have at least 8k spheres/core (maybe less for more compex shapes) mpi is not your friend. This in line with the estimate of 10k by Dion Weatherley (DEM8+beer)
- It looks like OpenMP sucks, but be aware that the benchmark script is heavily tuned for MPI. It includes huges verletDist and more time wasted on virtual interactions to minimize global updates.
- I believe tuning for OpenMP could make -j26 (or maybe 2xMPIx -j13) on par or faster than 26 MPI threads for less than a million particle. Given the additional difficulty, MPI's niche is for more than a million particles or more than one compute node.
- the nominal per-core throughput is not impressive. On an efficient script my laptop can approach 1e6Cu while we get 0.3e6Cu per core on Dahu. MPI is not to blame here, my laptop would also outperform Dahu on a single core.

5.3 Using YADE with cloud computing on Amazon EC2

(Note: we thank Robert Caulk for preparing and sharing this guide)

5.3.1 Summary

This guide is intended to help YADE users migrate their simulations to Amazon Web Service (AWS) EC2. Two of the most notable benefits of using scalable cloud computing for YADE include decreased upfront cost and increased productivity. The entire process, from launching an instance, to installing YADE, to running a YADE simulation on the cloud can be executed in under 5 minutes. Once the EC2 instance is running, you can submit YADE scripts the same way you would submit jobs on a local workstation.

5.3.2 Launching an EC2 instance

Start by signing into the console on Amazon EC2. This will require an existing or new Amazon account. Once you've signed in, you should find the EC2 console by clicking on 'services' in the upper left hand

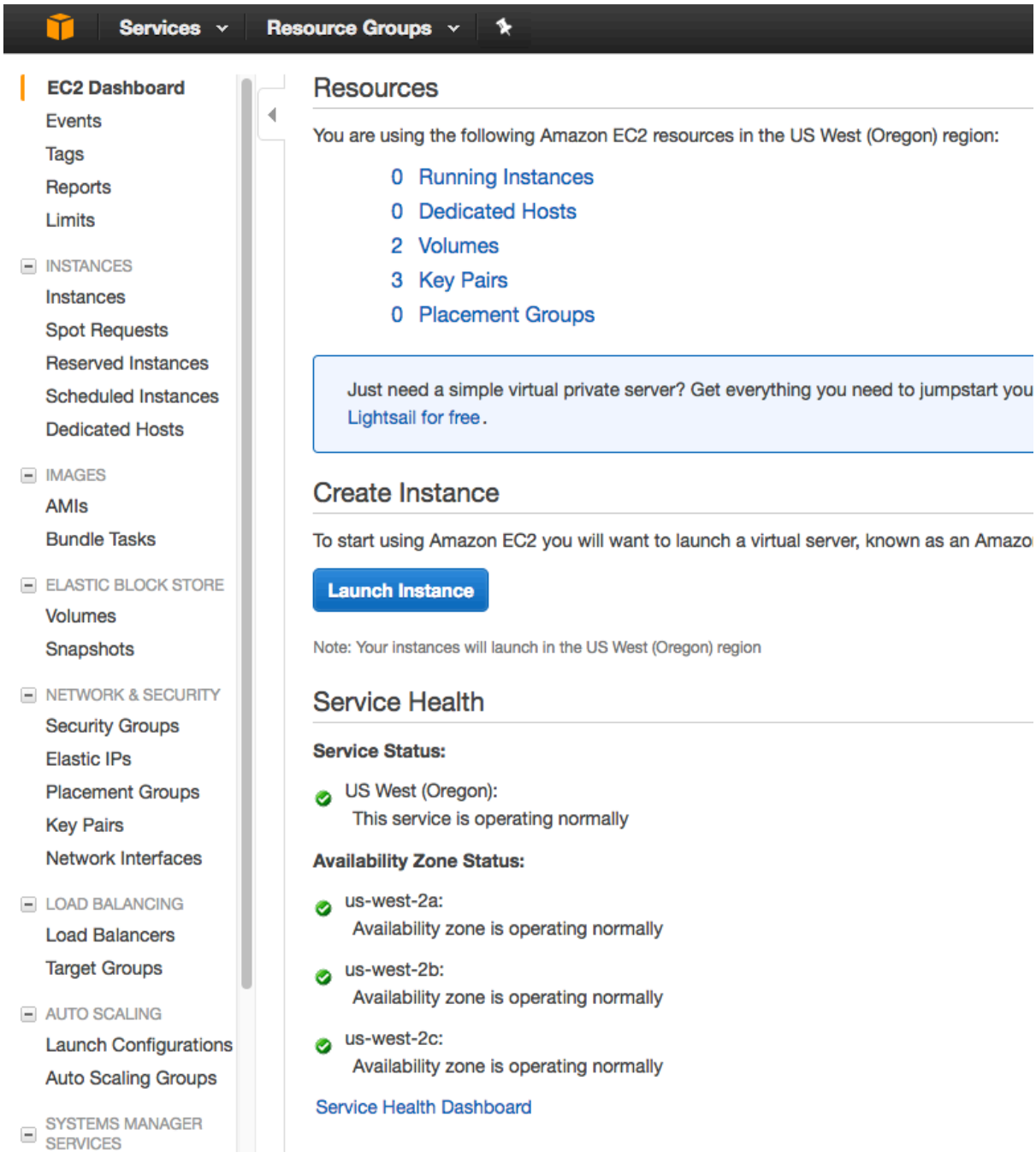


Fig. 2: Amazon Web Services (AWS) Console

corner of the AWS homepage. Start by clicking on the `launch an instance` blue button (Fig. *fig-console*). Select the Amazon Machine Image (AMI): `Ubuntu Server 16.04 LTS` (Fig. *fig-ubuntu*).

Fig. 3: Select Ubuntu server 16.04 LTS AMI

You will now select the instance type. It is worth looking at the specifications for each of the instances so you can properly select the power you need for you YADE simulation. This document will not go into detail in the selection of size, but you can find plenty of YADE specific performance reports that will help you decide. However, the instance type is an important selection. The `Compute Optimized` instances are necessary for most YADE simulations because they provide access to high performing processors and guaranteed computing power. The C3.2xlarge (Fig. *fig-type*) is equivalent to an 8 core 2.8ghz Xeon E5 with 25 mb of cache, which is likely the best option for medium-large scale YADE simulations.

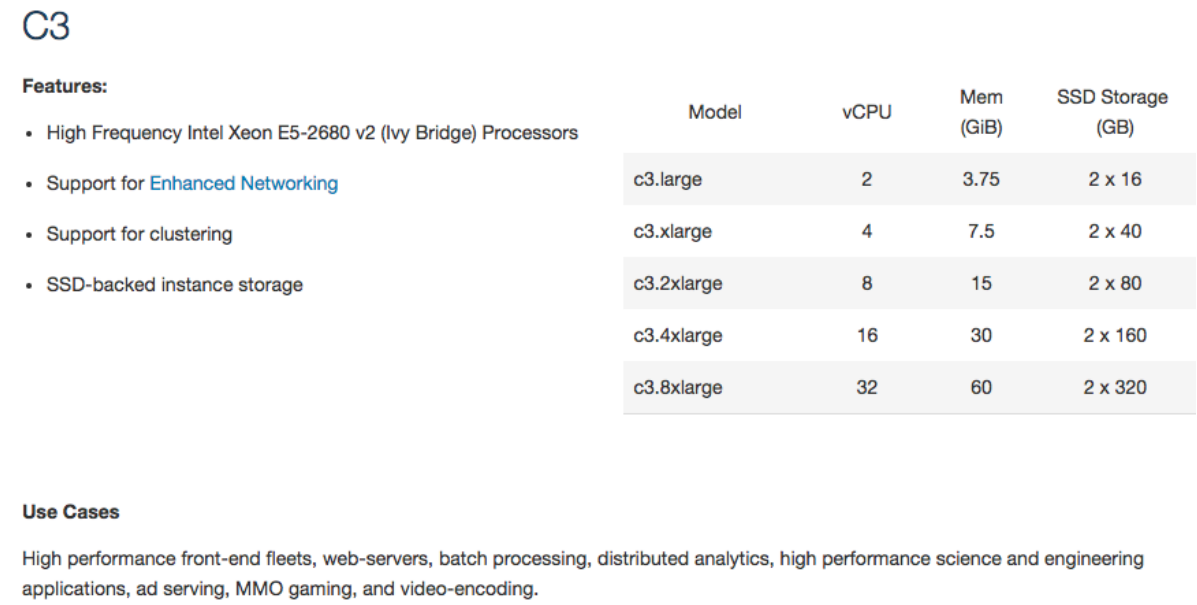


Fig. 4: Compute optimized (C3) instance tier

Before launching, you will be asked to `select an existing key pair or create a new key pair`. Create a new one, download it, and place it in a folder that you know the path to. Modify the permissions on the file by navigating to the same directory in the terminal and typing:

```
chmod 400 KeyPair.pem
```

Now the instance is launched, you will need to connect to it via SSH. On unix systems this is as easy as typing:

```
ssh -i path/to/KeyPair.pem ubuntu@ec2-XX-XXX-XX-XX.us-west-2.compute.amazon.com
```

into the terminal. There are other options such as using PuTTY, or even a java based terminal on the AWS website. You can find the necessary information by navigating to `Instances` in the left menu of the AWS console. Right click on the instance as shown in Fig. *fig-connect* and click connect.

You will be presented with the public DNS, which should look something like Fig. *fig-dns*.

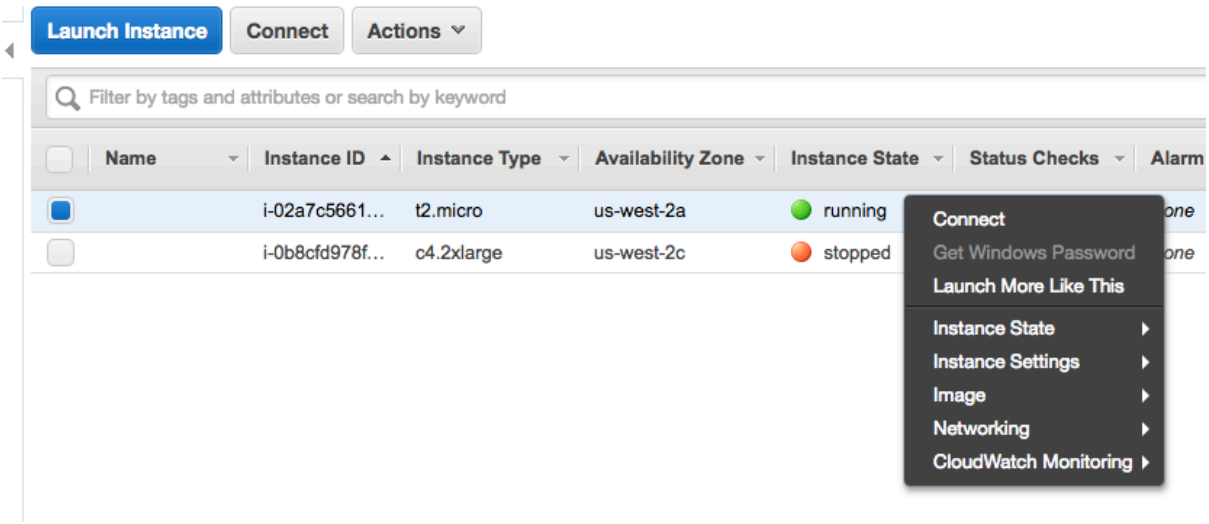


Fig. 5: Connecting to the instance

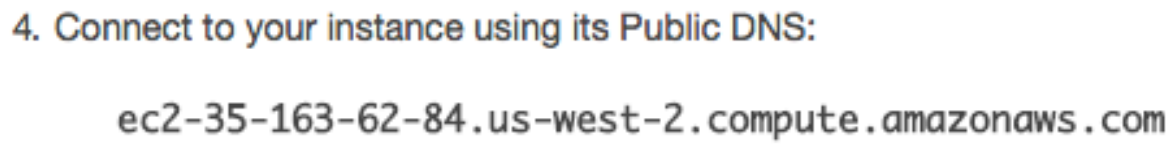


Fig. 6: Public DNS

5.3.3 Installing YADE and managing files

After you've connected to the instance through SSH, you will need to install YADE. The following commands should be issued to install yadedaily, python, and some other useful tools:

```
#install yadedaily
sudo bash -c 'echo "deb http://www.yade-dem.org/packages/ xenial/" >> /
↪etc/apt/sources.list'
wget -O - http://www.yade-dem.org/packages/yadedev_pub.gpg | sudo apt-key add -
sudo apt-get update
sudo apt-get install -y yadedaily

# install python
sudo apt-get -y install python
sudo apt-get -y install python-pip python-dev build-essential

# install htop
sudo apt-get -y install htop
```

Note that `..packages/ xenial/` should match the Ubuntu distribution. 16.04 LTS is Xenial, but if you chose to start Ubuntu 14.04, you will need to change 'xenial' to 'trusty'.

Finally, you will need to upload the necessary YADE files. If you have a folder with the contents of your simulation titled `yadeSimulation` you can upload the folder and its contents by issuing the following command:

```
scp -r -i path/to/KeyYADEbox.pem path/to/yadeSimulation ubuntu@ec2-XX-XXX-XX-XX.us-
↪west-2.compute.amazonaws.com:~/yadeSimulation
```

You should now be able to run your simulation by changing to the proper directory and typing:

```
yadedaily nameOfSimulation.py
```

In order to retrieve the output files (folder titled 'out' below) for post processing purposes, you will use the same command that you used to upload the folder, but the remote and local file destinations should be reversed:

```
scp -r -i path/to/KeyYADEbox.pem ubuntu@ec2-XX-XXX-XX-XX.us-west-2.compute.amazonaws.
↪com:~/yadeSimulation/out/ path/to/yadeSimulation/out
```

5.3.4 Plotting output in the terminal

One of the main issues encountered with cloud computing is the lack of graphical feedback. There is an easy solution for graphically checking the status of your simulations which makes use of gnuplot's wonderful 'terminal dumb' feature. Any data can be easily plotted by navigating to the subfolder where the simulation is saving its output and typing:

```
gnuplot
set terminal dumb
plot "data.txt" using 1:2 with lines
```

Where '1:2' refers to the columns in data.txt that you wish to plot against one another. Your output should look something like this:

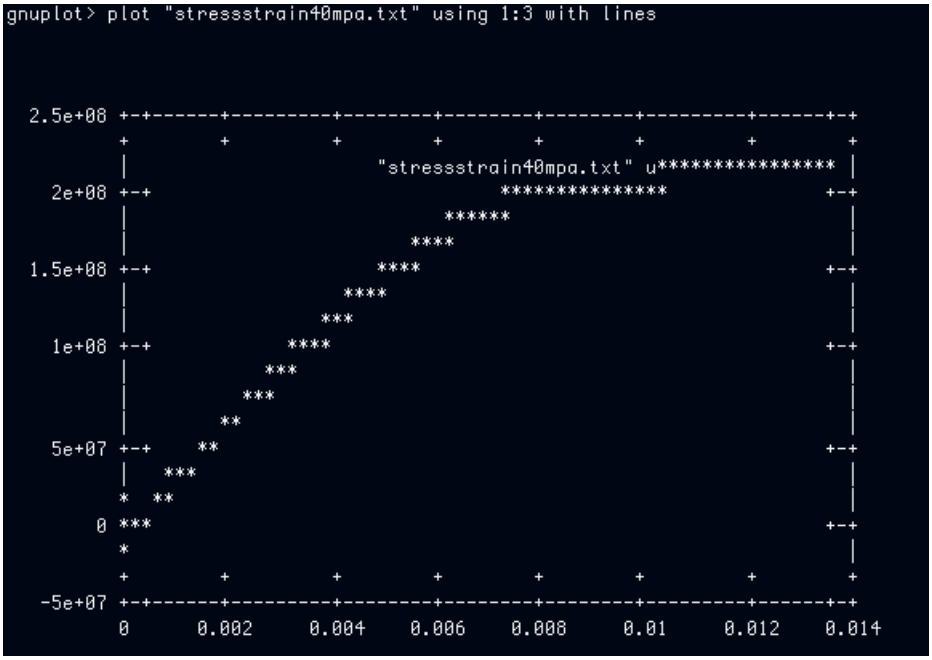


Fig. 7: Gnuplot output

5.3.5 Comments

- Amazon AWS allows you to stop your instance and restart it again later with the same files and package installations. If you wish to create several instances that all contain the same installation and file directory you can create a snapshot of your default image which you will be able to use to create various volumes that you can attach to new instances. These actions are all performed very easily and graphically through the EC2 console
- You can use Spot Instances, which are a special type of instance that allow you to bid on unused servers. The price is heavily discounted and worth looking into for any YADE user that wishes to run hundreds of hours of simulations.

- For most simulations, your computational efficiency will decrease if you use above 8 cores per simulation. It is preferable to use yadedaily-batch to distribute your cores accordingly so that you always dedicate 8 cores to each simulation and ensure 100% of the processor is running.
- Create a tmux session to avoid ending YADE simulations upon disconnecting from the server.

```
tmux  # starts a new session
tmux attach -t 0  # attach session 0
tmux kill -t 0  # kill session
## cntrl - b - d to move back to home
## cntrl - b - [ to navigate within the session
```

5.4 High precision calculations

Yade supports high and arbitrary precision `Real` type for performing calculations (see [Kozicki2022] for details). All tests and checks pass but still the current support is in testing phase. The backend library is boost multiprecision along with corresponding boost math toolkit.

The supported types are following:

type	bits	decimal places[1]	notes
`float`	32	6	hardware accelerated (not useful, it is only for testing purposes)
`double`	64	15	hardware accelerated
`long double`	80	18	hardware accelerated
`boost float128`	128	33	depending on processor type it may be hardware accelerated, wrapped by boost
`boost mpfr`	`Nbit`	`Nbit/(log(2)/log(10))`	uses external mpfr library, wrapped by boost
`boost cpp_bin_float`	`Nbit`	`Nbit/(log(2)/log(10))`	uses boost only, but is slower

The last two types are arbitrary precision, and their number of bits `Nbit` or decimal places is specified as argument during compilation.

Note

See file Real.hpp for details. All `Real` types pass the real type concept test from boost concepts. The support for Eigen and CGAL is done with numerical traits.

5.4.1 Installation

The *precompiled Yade Daily packages* for Ubuntu 22.04 and Debian Trixie are provided for high precision types `long double`, `float128` and `mpfr150`. To use high precision on other linux distributions Yade has to be compiled and installed from source code by following the regular *installation instructions*. With extra following caveats:

1. Following packages are required to be installed: `python3-mpmath libmpfr-dev libmpfrc++-dev libmpc-dev` (the `mpfr` and `mpc` related packages are necessary only to use `boost::multiprecision::mpfr` type). These packages are already listed in the *default requirements*.

[1] The amount of decimal places in this table is the amount of places which are completely determined by the binary represenation. *Few additional decimal digits* is necessary to fully reconstruct binary representation. A simple python example to demonstrate this fact: `for a in range(16): print(1./pow(2.,a))`, shows that every binary digit produces "extra" …25 at the end of decimal representation, but these decimal digits are not completely determined by the binary representation, because for example …37 is impossible to obtain there. More binary bits are necessary to represent …37, but the …25 was produced by the last available bit.

2. A g++ compiler version 9.2.1 or higher is required. It shall be noted that upgrading only the compiler on an existing linux installation (an older one, in which packages for different versions of gcc were not introduced) is difficult and it is not recommended. A simpler solution is to upgrade entire linux installation.

3. During cmake invocation specify:

    1. either number of bits as `REAL_PRECISION_BITS=.....`,

    2. or number of requested decimal places as `REAL_DECIMAL_PLACES=.....`, but not both

    3. optionally to use MPFR specify `ENABLE_MPFR=ON` (is `OFF` by default).

    4. optionally decide about using *quadruple, octuple or higher precisions* with `-DENABLE_MULTI_REAL_HP=ON` (default). This feature is independent of selecting the precision of `Real` type (in point 1. or 2. above) and works even when `Real` is chosen as `double` (i.e. no special choice is made: the default settings).

    The arbitrary precision (`mpfr` or `cpp_bin_float`) types are used only when more than 128 bits or more than 39 decimal places are requested. In such case if `ENABLE_MPFR=OFF` then the slower `cpp_bin_float` type is used. The difference in decimal places between 39 and 33 stems from the fact that 15 bits are used for exponent. Note: a fast quad-double (debian package `libqd-dev`) implementation with 62 decimal places is in the works with boost multiprecision team.

5.4.2 Supported modules

During *compilation* several Yade modules can be enabled or disabled by passing an `ENABLE_*` command line argument to cmake. The following table lists which modules are currently working with high precision (those marked with "maybe" were not tested):

ENABLE_* module name	HP support	cmake default setting	notes
ENABLE_GUI	yes	ON	native support[2]
ENABLE_CGAL	yes	ON	native support[2]
ENABLE_VTK	yes	ON	supported[3]
ENABLE_OPENMP	partial	ON	partial support[4]
ENABLE_MPI	maybe	OFF	not tested[5]
ENABLE_GTS	yes	ON	supported[6]
ENABLE_GL2PS	yes	ON	supported[Page 483, 6]
ENABLE_LINSOLV	no	OFF	not supported[7]
ENABLE_PARTIALSAT	no	OFF	not supported[Page 483, 7]
ENABLE_PFVFLOW	no	OFF	not supported[Page 483, 7]
ENABLE_TWOPHASEFLOW	no	OFF	not supported[Page 483, 7]
ENABLE_THERMAL	no	OFF	not supported[Page 483, 7]
ENABLE_LBMFLOW	yes	ON	supported[Page 483, 6]
ENABLE_SPH	maybe	OFF	not tested[8]
ENABLE_LIQMIGRATION	maybe	OFF	not tested[Page 483, 8]
ENABLE_MASK_ARBITRARY	maybe	OFF	not tested[Page 483, 8]
ENABLE_PROFILING	maybe	OFF	not tested[Page 483, 8]
ENABLE_POTENTIAL_BLOCKS	no	OFF	not supported[9]
ENABLE_POTENTIAL_PARTICLES	yes	ON	supported[10]
ENABLE_DEFORM	maybe	OFF	not tested[Page 483, 8]
ENABLE_OAR	maybe	OFF	not tested[Page 483, 8]
ENABLE_FEMLIKE	yes	ON	supported[Page 483, 6]
ENABLE_ASAN	yes	OFF	supported[Page 483, 6]
ENABLE_MPFR	yes	OFF	native support[2]
ENABLE_LS_DEM	no	ON	not supported[11]

[2] This feature is supported natively, which means that specific numerical traits were written for Eigen and for CGAL, as well as GUI and python support was added.
[3] VTK is supported via the compatibility layer which converts all numbers down to `double` type. See *below*.
[4] The OpenMPArrayAccumulator is experimentally supported for `long double` and `float128`. For types `mpfr` and `cpp_-`

The unsupported modules are automatically disabled during a high precision `cmake` stage.

5.4.3 Double, quadruple, octuple and higher precisions

Sometimes a critical section of the calculations in C++ would work better if it was performed in the higher precision to guarantee that it will produce the correct result in the default precision. A simple example is solving a system of linear equations (basically inverting a matrix) where some coefficients are very close to zero. Another example of alleviating such problem is the Kahan summation algorithm.

If *requirements* are satisfied, Yade supports higher precision multipliers in such a way that `RealHP<1>` is the `Real` type described above, and every higher number is a multiplier of the `Real` precision. `RealHP<2>` is double precision of `RealHP<1>`, `RealHP<4>` is quadruple precision and so on. The general formula for amount of decimal places is implemented in RealHP.hpp file and the number of decimal places used is simply a multiple N of decimal places in `Real` precision, it is used when native types are not available. The family of available native precision types is listed in the RealHPLadder type list.

All types listed in MathEigenTypes.hpp follow the same naming pattern: `Vector3rHP<1>` is the regular `Vector3r` and `Vector3rHP<N>` for any supported N uses the precision multiplier N. One could then use an Eigen algorithm for solving a system of linear equations with a higher N using `MatrixXrHP<N>` to obtain the result with higher precision. Then continuing calculations in default `Real` precision, after the critical section is done. The same naming convention is used for CGAL types, e.g. `CGAL_AABB_treeHP<N>` which are declared in file AliasCGAL.hpp.

Before we fully move to C++20 standard, one small restriction is in place: the precision multipliers actually supported are determined by these two defines in the RealHPConfig.hpp file:

1. `#define YADE_EIGENCGAL_HP (1)(2)(3)(4)(8)(10)(20)` - the multipliers listed here will work in C++ for `RealHP<N>` in CGAL and Eigen. They are cheap in compilation time, but have to be listed here nonetheless. After we move code to C++20 this define will be removed and all multipliers will be supported via single template constraint. This inconvenience arises from the fact that both CGAL and Eigen libraries offer template specializations only for a *specific* type, not a generalized family of types. Thus this define is used to declare the required template specializations.

Hint

The highest precision available by default N= `(20)` corresponds to 300 decimal places when compiling Yade with the default settings, without changing `REAL_DECIMAL_PLACES=……` cmake compilation option.

2. `#define YADE_MINIEIGEN_HP (1)(2)` - the precision multipliers listed here are exported to python, they are expensive: each one makes compilation longer by 1 minute. Adding more can be useful only for debugging purposes. The double `RealHP<2>` type is by default listed here to allow exploring the higher precision types from python. Also please note that `mpmath` supports only one precision at a time. Having different `mpmath` variables with different precision is poorly supported, albeit `mpmath` authors promise to improve that in the future. Fortunately this is not a big problem for Yade users because the general goal here is to allow more precise calculations in the critical sections of C++ code, not in python. This problem is partially mitigated by *changing* mpmath

`bin_float` the single-threaded version of accumulator is used. File lib/base/openmp-accu.hpp needs further testing. If in doubt, compile yade with `ENABLE_OPENMP=OFF`. In all other places OpenMP multithreading should work correctly.

[5] MPI support has not been tested and sending data over network hasn't been tested yet.

[6] The module was tested, the `yade --test` and `yade --check` pass, as well as most of examples are working. But it hasn't been tested extensively for all possible use cases.

[7] Not supported, the code uses external cholmod library which supports only `double` type. To make it work a native Eigen solver for linear equations should be used.

[8] This feature is `OFF` by default, the support of this feature has not been tested.

[9] Potential blocks use external library coinor for linear programming, this library uses `double` type only. To make it work a linear programming routine has to be implemented using Eigen or coinor library should start using C++ templates or a converter/wrapper similar to LAPACK library should be used.

[10] The module is enabled by default, the `yade --test` and `yade --check` pass, as well as most of examples are working. However the calculations are performed at lower `double` precision. A wrapper/converter layer for LAPACK library has been implemented. To make it work with full precision these routines should be reimplemented using Eigen.

[11] Possible future enchancement. See comments there .

precision each time when a C++ python conversion occurs. So one should keep in mind that the variable mpmath.mp.dps always reflects the precision used by latest conversion performed, even if that conversion took place in GUI (not in the running script). Existing mpmath variables are not truncated to lower precision, their extra digits are simply ignored until mpmath.mp.dps is increased again, however the truncation might occur during assignment.

On some occasions it is useful to have an intuitive up-conversion between C++ types of different precisions, say for example to add RealHP<1> to RealHP<2> type. The file UpconversionOfBasicOperatorsHP.hpp serves this purpose. This header is not included by default, because more often than not, adding such two different types will be a mistake (efficiency–wise) and compiler will catch them and complain. After including this header this operation will become possible and the resultant type of such operation will be always the higher precision of the two types used. This file should be included only in .cpp files. If it was included in any .hpp file then it could pose problems with C++ type safety and will have unexpected consequences. An example usage of this header is in the following test routine.

Warning

Trying to use N unregistered in YADE_MINIEIGEN_HP for a Vector3rHP<N> type inside the YADE_CLASS_BASE_DOC_ATTRS_* macro to export it to python will not work. Only these N listed in YADE_MINIEIGEN_HP will work. However it is safe (and intended) to use these from YADE_EIGENCGAL_HP in the C++ calculations in critical sections of code, without exporting them to python.

5.4.4 Compatibility

Python

To declare python variables with Real and RealHP<N> precision use functions *math.Real(…)*, *math.Real1(…)*, *math.Real2(…)*. Supported are precisions listed in YADE_MINIEIGEN_HP, but please note the *mpmath-conversion-restrictions*.

Python has native support for high precision types using mpmath package. Old Yade scripts that use *supported modules* can be immediately converted to high precision by switching to yade.minieigenHP. In order to do so, the following line:

```
from minieigen import *
```

has to be replaced with:

```
from yade.minieigenHP import *
```

Respectively import minieigen has to be replaced with import yade.minieigenHP as minieigen, the old name as minieigen being used only for the sake of backward compatibility. Then high precision (binary compatible) version of minieigen is used when non double type is used as Real.

The RealHP<N> *higher precision* vectors and matrices can be accessed in python by using the .HPn module scope. For example:

```
import yade.minieigenHP as mne
mne.HP2.Vector3(1,2,3) # produces Vector3 using RealHP<2> precision
mne.Vector3(1,2,3)     # without using HPn module scope it defaults to RealHP<1>
```

The respective math functions such as:

```
import yade.math as mth
mth.HP2.sqrt(2) # produces square root of 2 using RealHP<2> precision
mth.sqrt(2)     # without using HPn module scope it defaults to RealHP<1>
```

are supported as well and work by using the respective C++ function calls, which is usually faster than the mpmath functions.

Warning

There may be still some parts of python code that were not migrated to high precision and may not work well with `mpmath` module. See *debugging section* for details.

C++

Before introducing high precision it was assumed that `Real` is actually a POD `double` type. It was possible to use `memset(…)`, `memcpy(…)` and similar functions on `double`. This was not a good approach and even some compiler `#pragma` commands were used to silence the compilation warnings. To make `Real` work with other types, this assumption had to be removed. A single `memcpy(…)` still remains in file openmp-accu.hpp and will have to be removed. In future development such raw memory access functions are to be avoided.

All remaining `double` were replaced with `Real` and any attempts to use `double` type in the code will fail in the gitlab-CI pipeline.

Mathematical functions of all high precision types are wrapped using file MathFunctions.hpp, these are the inline redirections to respective functions of the type that Yade is currently being compiled with. The code will not pass the pipeline checks if `std::` is used. All functions that take `Real` argument should now call these functions in `yade::math::` namespace. Functions which take *only* `Real` arguments may omit `math::` specifier and use ADL instead. Examples:

1. Call to `std::min(a,b)` is replaced with `math::min(a,b)`, because `a` or `b` may be `int` (non `Real`) therefore `math::` is necessary.
2. Call to `std::sqrt(a)` can be replaced with either `sqrt(a)` or `math::sqrt(a)` thanks to ADL, because `a` is always `Real`.

If a new mathematical function is needed it has to be added in the following places:

1. lib/high-precision/MathFunctions.hpp or lib/high-precision/MathComplexFunctions.hpp or lib/high-precision/MathSpecialFunctions.hpp, depending on function type.
2. py/high-precision/_math.cpp, see *math module* for details.
3. py/tests/testMath.py
4. py/tests/testMathHelper.py

The tests for a new function are to be added in py/tests/testMath.py in one of these functions: `oneArgMathCheck(…):`, `twoArgMathCheck(…):`, `threeArgMathCheck(…):`. A table of approximate expected error tolerances in `self.defaultTolerances` is to be supplemented as well. To determine tolerances with better confidence it is recommended to temporarily increase number of tests in the test loop. To determine tolerances for currently implemented functions a `range(1000000)` in the loop was used.

Note

When passing arguments in `C++` in function calls it is preferred to use `const Real&` rather than to make a copy of the argument as `Real`. The reason is following: in non high-precision regular case both the `double` type and the reference have 8 bytes. However `float128` is 16 bytes large, while its reference is still only 8 bytes. So for regular precision, there is no difference. For all higher precision types it is beneficial to use `const Real&` as the function argument. Also for `const Vector3r&` arguments the speed gain is larger, even without high precision.

Using higher precisions in C++

As mentioned above `RealHP<1>` is the `Real` type and every higher number is a multiplier of the `Real` precision. `RealHP<2>` is twice the precision of `RealHP<1>`, `RealHP<4>` is quadruple precision and so on. In C++ you have access to these higher precision typedefs at all time, so it is possible to write some critical

part of an algorithm in higher precision by declaring the respective variables to be of type `RealHP<2>` or `RealHP<4>` or higher.

String conversions

On the `python` side it is recommended to use *math.Real(…)* *math.Real1(…)*, or *math.toHP1(…)* to declare `python` variables and *math.radiansHP1(…)* to convert angles to radians using *full Pi precision*.

On the `C++` side it is recommended to use yade::math::toString(…) and yade::math::fromStringReal(…) conversion functions instead of `boost::lexical_cast<std::string>(…)`. The `toString` and its high precision version toStringHP functions (in file RealIO.hpp) guarantee full precision during conversion. It is important to note that `std::to_string` does not guarantee this and `boost::lexical_cast` does not guarantee this either.

For higher precision types it is possible to control in runtime the precision of `C++` `python` during the `RealHP<N>` string conversion by changing the *math.RealHPConfig.extraStringDigits10* static parameter. Each decimal digit needs $\log_{10}(2) \approx 3.3219$ bits. The `std::numeric_limits<Real>::digits10` provides information about how many decimal digits are completely determined by binary representation, meaning that these digits are absolutely correct. However to convert back to binary more decimal digits are necessary because $\log_2(10) \approx 0.3010299$ decimal digits are used by each bit, and the last digit from `std::numeric_limits<Real>::digits10` is not sufficient. In general 3 or more in *extraStringDigits10* is enough to have an always working number round tripping. However if one wants to only extract results from python, without feeding them back in to continue calculations then a smaller value of *extraStringDigits10* is recommended, like 0 or 1, to avoid a fake sense of having more precision, when it's not there: these extra decimal digits are not correct in decimal sense. They are only there to have working number round tripping. See also a short discussion about this with boost developers. Also see file RealHPConfig.cpp for more details.

Note

The parameter `extraStringDigits10` does not affect `double` conversions, because `boost::python` uses an internal converter for this particular type. It might be changed in the future if the need arises. E.g. using a class similar to ThinRealWrapper.

It is important to note that creating higher types such as `RealHP<2>` from string representation of `RealHP<1>` is ambiguous. Consider following example:

```
import yade.math as mth

mth.HP1.getDecomposedReal(1.23)['bits']
Out[2]: '10011101011100001010001111010111000010100011110101110'

mth.HP2.getDecomposedReal('1.23')['bits']  # passing the same arg in decimal format␣
↪to HP2 produces nonzero bits after the first 53 bits of HP1
Out[3]:
↪'10011101011100001010001111010111000010100011110101110000101000111101011100001
↪'

mth.HP2.getDecomposedReal(mth.HP1.toHP2(1.23))['bits'] # it is possible to use␣
↪yade.math.HPn.toHPm(…) conversion, which preserves binary representation
Out[4]:
↪'10011101011100001010001111010111000010100011110101110000000000000000000000000000000000000000000000000
↪'
```

Which of these two `RealHP<2>` binary representations is more desirable depends on what is needed:

1. The best binary approximation of a `1.23` decimal.
2. Reproducing the 53 binary bits of that number into a higher precision to continue the calculations on **the same** number which was previously in lower precision.

To achieve 1. simply pass the argument `'1.23'` as string. To achieve 2. use *math.HPn.toHPm(…)* or *math.Realn(…)* conversion, which maintains binary fidelity using a single static_cast<RealHP<m>>(…). Similar problem is discussed in mpmath and boost documentation.

The difference between *toHPn* and *Realn* is following: the functions `HPn.toHPm` create a $\mathtt{m} \times \mathtt{n}$ matrix converting from `RealHP<n>` to `RealHP<m>`. When $\mathtt{n} < \mathtt{m}$ then extra bits are set to zero (case 2 above, depending on what is required one might say that "precision loss occurs"). The functions *math.Real(…)*, *math.Real1(…)*, *math.Real2(…)* are aliases to the diagonal of this matrix (case 1 above, depending on what is required one might say that "no conversion loss occurs" when using them).

Hint

All `RealHP<N>` function arguments that are of type higher than `double` can also accept decimal strings. This allows to preserve precision above python default floating point precision.

Warning

On the contrary all the function arguments that are of type `double` can not accept decimal strings. To mitigate that one can use `toHPn(…)` converters with string arguments.

Hint

To make debugging of this problem easier the function *math.toHP1(…)* will raise RuntimeError if the argument is a python float (not a decimal string).

Warning

I cannot stress this problem enough, please try running `yade --check` (or `yade ./checkGravityRungeKuttaCashKarp54.py`) in precision different than `double` after changing this line into `g = -9.81`. In this (particular and simple) case the `getCurrentPos()` function fails on the python side because low-precision `g` is multiplied by high-precision `t`.

Complex types

Complex numbers are supported as well. All standard `C++` functions are available in lib/high-precision/MathComplexFunctions.hpp and also are exported to python in py/high-precision/_math.cpp. There is a cmake compilation option `ENABLE_COMPLEX_MP` which enables using better complex types from `boost::multiprecision` library for representing `ComplexHP<N>` family of types: `complex128`, `mpc_complex`, `cpp_complex` and `complex_adaptor`. It is ON by default whenever possible: for boost version >= 1.71. For older boost the `ComplexHP<N>` types are represented by `std::complex<RealHP<N>>` instead, which has larger numerical errors in some mathematical functions.

When using the `ENABLE_COMPLEX_MP=ON` (default) the previously mentioned lib/high-precision/UpconversionOfBasicOperatorsHP.hpp is not functional for complex types, it is a reported problem with the boost library.

When using MPFR type, the `libmpc-dev` package has to be installed (mentioned above).

Eigen and CGAL

Eigen and CGAL libraries have native high precision support.

- All declarations required by Eigen are provided in files EigenNumTraits.hpp and MathEigenTypes.hpp

- All declarations required by CGAL are provided in files CgalNumTraits.hpp and AliasCGAL.hpp

VTK

Since VTK is only used to record results for later viewing in other software, such as paraview, the recording of all decimal places does not seem to be necessary (for now). Hence all recording commands in `C++` convert `Real` type down to `double` using `static_cast<double>` command. This has been implemented via classes `vtkPointsReal`, `vtkTransformReal` and `vtkDoubleArrayFromReal` in file VTKCompatibility.hpp. Maybe VTK in the future will support non `double` types. If that will be needed, the interface can be updated there.

LAPACK

Lapack is an external library which only supports `double` type. Since it is not templatized it is not possible to use it with `Real` type. Current solution is to down-convert arguments to `double` upon calling linear equation solver (and other functions), then convert them back to `Real`. This temporary solution omits all benefits of high precision, so in the future Lapack is to be replaced with Eigen or other templatized libraries which support arbitrary floating point types.

5.4.5 Debugging

High precision is still in the experimental stages of implementation. Some errors may occur during use. Not all of these errors are caught by the checks and tests. Following examples may be instructive:

1. Trying to use const references to Vector3r members - a type of problem with results in a segmentation fault during runtime.
2. A part of python code does not cooperate with mpmath - the checks and tests do not cover all lines of the python code (yet), so more errors like this one are expected. The solution is to put the non compliant python functions into py/high-precision/math.py. Then replace original calls to this function with function in `yade.math`, e.g. `numpy.linspace(…)` is replaced with `yade.math.linspace(…)`.

The most flexibility in debugging is with the `long double` type, because special files ThinRealWrapper.hpp, ThinComplexWrapper.hpp were written for that. They are implemented with boost::operators, using partially ordered field. Note that they do not provide operator++.

A couple of `#defines` were introduced in these two files to help debugging more difficult problems:

1. `YADE_IGNORE_IEEE_INFINITY_NAN` - it can be used to detect all occurrences when `NaN` or `Inf` are used. Also it is recommended to use this define when compiling Yade with `-Ofast` flag, without `-fno-associative-math -fno-finite-math-only -fsigned-zeros`
2. `YADE_WRAPPER_THROW_ON_NAN_INF_REAL`, `YADE_WRAPPER_THROW_ON_NAN_INF_COMPLEX` - can be useful for debugging when calculations go all wrong for unknown reason.

Also refer to *address sanitizer section*, as it is most useful for debugging in many cases.

Hint

If crash is inside a macro, for example `YADE_CLASS_BASE_DOC_ATTRS_CTOR_PY`, it is useful to know where inside this macro the problem happens. For this purpose it is possible to use `g++` preprocessor to remove the macro and then compile the postprocessed code without the macro. Invoke the preprocessor with some variation of this command:

```
g++ -E -P core/Body.hpp -I ./ -I /usr/include/eigen3 -I /usr/include/python3.7m > /
↪tmp/Body.hpp
```

Maybe use clang-format so that this file is more readable:

```
./scripts/clang-formatter.sh /tmp/Body.hpp
```

Be careful because such files tend to be large and clang-format is slow. So sometimes it is more useful to only use the last part of the file, where the macro was postprocessed. Then replace the macro in the original file in question, and then continue debugging. But this time it will be revealed where inside a macro the problem occurs.

Note

When *asking questions* about High Precision it is recommended to start the question title with `[RealHP]`.

Chapter 6

Short-courses

6.1 THM short-course

This tutorial was used by Bruno Chareyre and Robert Caulk to help teach the 3-day Thermo-Hydro-Mechanical coupling short-course in Amsterdam on June 20, 2022.

Slides and other supplementary material can be downloaded here.

Meanwhile, the following hands-on guides are designed to be followed sequentially by someone who is unfamiliar with Python and Yade:

6.1.1 Installing Yade (for Windows and Mac users)

In preparation for the THM short-course, we ask the participants to have a Linux Debian distribution installed on their laptop prior to arrival.

If you already have a debian distribution on your laptop, please follow the installation instructions on our website.

If you do not have a debian distribution on your laptop, you have three ways to get one:

Easiest way - Use our premade Virtual Machine (Windows)

We created a full debian machine preloaded with Yade + Paraview + Kate. You can install this easily with the following steps:

1. Download and install VirtualBox for your OS.
2. Download this yade_machine.ova file (this step may take a few minutes, so please be patient. The file is 6 gb.)
3. Open VirtualBox and click “Tools>Import”
4. Locate the ‘yade_machine.ova’ file that you downloaded, and click “Next”
5. Edit the system properties to suit your needs. Set the CPU count to half of your laptop CPUs and the RAM to half of your total laptop RAM.
6. Click “Import” on bottom right.
7. Start the machine and it should bring you into the Ubuntu desktop where you can open a new terminal (Ctrl-alt-T) and type

```
yadedaily --version
```

To test that yade is already installed and ready to go.

Login details (feel free to change these as soon as you are into your new VM): user: yadeuser password: yadeuser

Less easy way - Create your own Virtual Machine (MacOS)

This is if the direction above do not work for you. The end result is the same.

1. Download and install VirtualBox for your OS
2. Download the Ubuntu 20.04 image
3. Open VirtualBox and select "Machine>New..."
4. Select "Type" as "Linux" and "Version" as "Ubuntu (64-bit)" (If you do not see Ubuntu 64-bit, contact me directly for assistance).
5. Select at least 4gb of ram (preferably 8gb), Select "Create a virtual hard disk now"
6. Name the machine and then click "Create"
7. Choose 20-30gb of storage, leave the remaining options as default.
8. Click "create"
9. Click "Start" and then find the Ubuntu 20.04 .iso that you downloaded from Step 2.
10. Follow the installation instructions (this will take some time depending on your HDD speed)
11. Once Ubuntu is fully installed and you are inside the machine, go ahead and install yade by opening a terminal (Ctrl-alt-T to open a new terminal).

```
sudo bash -c 'echo "deb http://www.yade-dem.org/packages/ focal main" >> /
↪etc/apt/sources.list.d/yadedaily.list'
wget -O - http://www.yade-dem.org/packages/yadedev_pub.gpg | sudo tee /
↪etc/apt/trusted.gpg.d/yadedaily.asc
sudo apt-get update
sudo apt-get install yadedaily
```

1. Download and install Paraview.

You are all set!

Hard way - Dual Boot (MacOS or Windows)

Find instructions here

6.1.2 Introduction to Bash and Python

Terminal

The terminal is a shell designed to let us interact with the computer and its filing system with a basic set of commands:

```
user@machine:~$                          # user operating at machine, in the directory␣
↪~ (= user's home directory)
user@machine:~$ ls .                     # list contents of the current directory
user@machine:~$ ls foo                   # list contents of directory foo, relative to␣
↪the dcurrent directory ~ (= ls ~/foo = ls /home/user/foo)
user@machine:~$ ls /tmp                  # list contents of /tmp
user@machine:~$ cd foo                   # change directory to foo
user@machine:~/foo$ ls ~                 # list home directory (= ls /home/user)
user@machine:~/foo$ cd bar               # change to bar (= cd ~/foo/bar)
user@machine:~/foo/bar$ cd ../../foo2    # go to the parent directory twice, then to␣
↪foo2 (cd ~/foo/bar/../../foo2 = cd ~/foo2 = cd /home/user/foo2)
user@machine:~/foo2$ cd                  # go to the home directory (= ls ~ = ls /
↪home/user)
user@machine:~$
```

Keys

Useful keys on the command-line are:

<tab>	show possible completions of what is being typed (use abundantly!)
^C (=Ctrl+C)	delete current line
^D	exit the shell
↑↓	move up and down in the command history
^C	interrupt currently running program
^\	kill currently running program
Shift-PgUp	scroll the screen up (show past output)
Shift-PgDown	scroll the screen down (show future output; works only on quantum computers)

Starting yade

If yade is installed on the machine, it can be (roughly speaking) run as any other program; without any arguments, it runs in the "dialog mode", where a command-line is presented:

```
user@machine:~$ yade
Welcome to Yade 2022.01a
TCP python prompt on localhost:9002, auth cookie `adcusk'
XMLRPC info provider on http://localhost:21002
[[ ^L clears screen, ^U kills line. F12 controller, F11 3d view, F10 both, F9␣
 ↪generator, F8 plot. ]]
Yade [1]:                                          #### hit ^D to exit
Do you really want to exit ([y]/n)?
Yade: normal exit.
```

The command-line is in fact `python`, enriched with some yade-specific features. (Pure python interpreter can be run with `python` or `ipython` commands).

Instead of typing commands one-by-one on the command line, they can be be written in a file (with the .py extension) and given as argument to Yade:

```
user@machine:~$ yade simulation.py
```

For a complete help, see `man yade`

Exercises

1. Open the terminal, navigate to your home directory
2. Create a new empty file and save it in `~/first.py`
3. Change directory to `/tmp`; delete the file `~/first.py`
4. Run program `xeyes`
5. Look at the help of Yade.
6. Look at the *manual page* of Yade
7. Run Yade, exit and run it again.

Yade basics

Yade objects are constructed in the following manner (this process is also called "instantiation", since we create concrete instances of abstract classes: one individual sphere is an instance of the abstract *Sphere*, like Socrates is an instance of "man"):

```
Yade [1]: Sphere            # try also Sphere?
Out[1]: yade.wrapper.Sphere

Yade [2]: s=Sphere()        # create a Sphere, without specifying any attributes

Yade [3]: s.radius          # 'nan' is a special value meaning "not a number" (i.e.␣
↪not defined)
Out[3]: nan

Yade [4]: s.radius=2        # set radius of an existing object

Yade [5]: s.radius
Out[5]: 2.0

Yade [6]: ss=Sphere(radius=3)    # create Sphere, giving radius directly

Yade [7]: s.radius, ss.radius     # also try typing s.<tab> to see defined attributes
Out[7]: (2.0, 3.0)
```

Particles

Particles are the "data" component of simulation; they are the objects that will undergo some processes, though do not define those processes yet.

Singles

There is a number of pre-defined functions to create particles of certain type; in order to create a sphere, one has to (see the source of *utils.sphere* for instance):

1. Create *Body*
2. Set *Body.shape* to be an instance of *Sphere* with some given radius
3. Set *Body.material* (last-defined material is used, otherwise a default material is created)
4. Set position and orientation in *Body.state*, compute mass and moment of inertia based on *Material* and *Shape*

In order to avoid such tasks, shorthand functions are defined in the *utils* module; to mention a few of them, they are *utils.sphere*, *utils.facet*, *utils.wall*.

```
Yade [8]: s=utils.sphere((0,0,0),radius=1)    # create sphere particle centered at␣
↪(0,0,0) with radius=1

Yade [9]: s.shape                        # s.shape describes the geometry of the␣
↪particle
Out[9]: <Sphere instance at 0x4a38900>

Yade [10]: s.shape.radius                 # we already know the Sphere class
Out[10]: 1.0

Yade [11]: s.state.mass, s.state.inertia # inertia is computed from density and␣
↪geometry
Out[11]:
(4188.790204786391,
 Vector3(1675.516081914556253,1675.516081914556253,1675.516081914556253))

Yade [12]: s.state.pos                    # position is the one we prescribed
Out[12]: Vector3(0,0,0)
```

(continues on next page)

(continued from previous page)

```
Yade [13]: s2=utils.sphere((-2,0,0),radius=1,fixed=True)      # explanation below
```

In the last example, the particle was fixed in space by the `fixed=True` parameter to *utils.sphere*; such a particle will not move, creating a primitive boundary condition.

A particle object is not yet part of the simulation; in order to do so, a special function *O.bodies.append* (also see *Omega::bodies* and *Scene*) is called:

```
Yade [14]: O.bodies.append(s)             # adds particle s to the simulation; returns␣
 ↪id of the particle(s) added
Out[14]: 0
```

Packs

There are functions to generate a specific arrangement of particles in the *pack* module; for instance, cloud (random loose packing) of spheres can be generated with the *pack.SpherePack* class:

```
Yade [15]: from yade import pack

Yade [16]: sp=pack.SpherePack()                    # create an empty cloud; SpherePack␣
 ↪contains only geometrical information

Yade [17]: sp.makeCloud((1,1,1),(2,2,2),rMean=.2) # put spheres with defined radius␣
 ↪inside box given by corners (1,1,1) and (2,2,2)
Out[17]: 7

Yade [18]: for c,r in sp: print(c,r)                # print center and radius of all␣
 ↪particles (SpherePack is a sequence which can be iterated over)
   ....:
Vector3(1.447128180986672508,1.435608208977069422,1.493787586742612028) 0.2
Vector3(1.278079860012052649,1.721794600516157292,1.215656167496782292) 0.2
Vector3(1.79585248666189301,1.513828490459161236,1.679664652099744693) 0.2
Vector3(1.710406847877673275,1.77861050366229434,1.318800371235737678) 0.2
Vector3(1.783536233716795705,1.253677332320032534,1.305869452369571082) 0.2
Vector3(1.225738759737205585,1.667235685677142198,1.799579103174806072) 0.2
Vector3(1.263376963390541396,1.209285664294037943,1.219062597235276435) 0.2

Yade [19]: sp.toSimulation()                       # create particles and add them to␣
 ↪the simulation
Out[19]: [1, 2, 3, 4, 5, 6, 7]
```

Boundaries

utils.facet (triangle *Facet*) and *utils.wall* (infinite axes-aligned plane *Wall*) geometries are typically used to define boundaries. For instance, a "floor" for the simulation can be created like this:

```
Yade [20]: O.bodies.append(utils.wall(-1,axis=2))
Out[20]: 8
```

There are other conveinence functions (like *utils.facetBox* for creating closed or open rectangular box, or family of *ymport* functions)

Look inside

The simulation can be inspected in several ways. All data can be accessed from python directly:

```
Yade [21]: len(O.bodies)
Out[21]: 9

Yade [22]: O.bodies[10].shape.radius   # radius of body #10 (will give error if not␣
↪sphere, since only spheres have radius defined)
[31m---------------------------------------------------------------------------[39m
[31mIndexError[39m                                Traceback (most recent call last)
[36mCell[39m[36m [39m[32mIn[22][39m[32m, line 1[39m
[32m----> [39m[32m1[39m␣
↪[43mO[49m[43m.[49m[43mbodies[49m[43m[[49m[32;43m10[39;49m[43m][49m.shape.radius  ␣
↪[38;5;66;03m# radius of body #10 (will give error if not sphere, since only spheres␣
↪have radius defined)[39;00m

[31mIndexError[39m: Body id out of range.

Yade [23]: O.bodies[12].state.pos      # position of body #12
[31m---------------------------------------------------------------------------[39m
[31mIndexError[39m                                Traceback (most recent call last)
[36mCell[39m[36m [39m[32mIn[23][39m[32m, line 1[39m
[32m----> [39m[32m1[39m␣
↪[43mO[49m[43m.[49m[43mbodies[49m[43m[[49m[32;43m12[39;49m[43m][49m.state.pos     ␣
↪[38;5;66;03m# position of body #12[39;00m

[31mIndexError[39m: Body id out of range.
```

Besides that, Yade says this at startup (the line preceding the command-line):

```
[[ ^L clears screen, ^U kills line. F12 controller, F11 3d view, F10 both, F9␣
↪generator, F8 plot. ]]
```

Controller
: Pressing `F12` brings up a window for controlling the simulation. Although typically no human intervention is done in large simulations (which run "headless", without any graphical interaction), it can be handy in small examples. There are basic information on the simulation (will be used later).

3d view
: The 3d view can be opened with F11 (or by clicking on button in the *Controller* – see below). There is a number of keyboard shortcuts to manipulate it (press `h` to get basic help), and it can be moved, rotated and zoomed using mouse. Display-related settings can be set in the "Display" tab of the controller (such as whether particles are drawn).

Inspector
: *Inspector* is opened by clicking on the appropriate button in the *Controller*. It shows (and updates) internal data of the current simulation. In particular, one can have a look at engines, particles (*Bodies*) and interactions (*Interactions*). Clicking at each of the attribute names links to the appropriate section in the documentation.

Engines

Engines define processes undertaken by particles. As we know from the theoretical introduction, the sequence of engines is called *simulation loop*. Let us define a simple interaction loop:

```
Yade [24]: O.engines=[                    # newlines and indentations are not␣
↪important until the brace is closed
   ....:     ForceResetter(),
```

(continues on next page)

(continued from previous page)

```
   ....:     InsertionSortCollider([Bo1_Sphere_Aabb(),Bo1_Wall_Aabb()]),
   ....:     InteractionLoop(           # dtto for the parenthesis here
   ....:         [Ig2_Sphere_Sphere_ScGeom(),Ig2_Wall_Sphere_ScGeom()],
   ....:         [Ip2_FrictMat_FrictMat_FrictPhys()],
   ....:         [Law2_ScGeom_FrictPhys_CundallStrack()]
   ....:     ),
   ....:     NewtonIntegrator(damping=.2,label='newtonCustomLabel')      # define a␣
↪label newtonCustomLabel under which we can access this engine easily
   ....: ]
   ....:

Yade [25]: O.engines
Out[25]:
[<ForceResetter instance at 0x49efc30>,
 <InsertionSortCollider instance at 0x4acdb10>,
 <InteractionLoop instance at 0x48a1f00>,
 <NewtonIntegrator instance at 0x3c03e90>]

Yade [26]: O.engines[-1]==newtonCustomLabel    # is it the same object?
Out[26]: True

Yade [27]: newtonCustomLabel.damping
Out[27]: 0.2
```

Instead of typing everything into the command-line, one can describe simulation in a file (*script*) and then run yade with that file as an argument. We will therefore no longer show the command-line unless necessary; instead, only the script part will be shown. Like this:

```
O.engines=[                    # newlines and indentations are not important until the␣
↪brace is closed
        ForceResetter(),
        InsertionSortCollider([Bo1_Sphere_Aabb(),Bo1_Wall_Aabb()]),
        InteractionLoop(           # dtto for the parenthesis here
                [Ig2_Sphere_Sphere_ScGeom(),Ig2_Wall_Sphere_ScGeom()],
                [Ip2_FrictMat_FrictMat_FrictPhys()],
                [Law2_ScGeom_FrictPhys_CundallStrack()]
        ),
        GravityEngine(gravity=(0,0,-9.81)),                    # 9.81 is the gravity␣
↪acceleration, and we say that
        NewtonIntegrator(damping=.2,label='newtonCustomLabel') # define a label under␣
↪which we can access this engine easily
]
```

Besides engines being run, it is likewise important to define how often they will run. Some engines can run only sometimes (we will see this later), while most of them will run always; the time between two successive runs of engines is *timestep* (Δt). There is a mathematical limit on the timestep value, called *critical timestep*, which is computed from properties of particles. Since there is a function for that, we can just set timestep using *utils.PWaveTimeStep*:

```
O.dt=utils.PWaveTimeStep()
```

Each time when the simulation loop finishes, time `O.time` is advanced by the timestep `O.dt`:

```
Yade [28]: O.dt=0.01

Yade [29]: O.time
Out[29]: 0.0
```

(continues on next page)

(continued from previous page)

```
Yade [30]: O.step()

Yade [31]: O.time
Out[31]: 0.01
```

For experimenting with a single simulations, it is handy to save it to memory; this can be achieved, once everything is defined, with:

```
O.saveTmp()
```

6.1.3 Day 1 - Yade Hands-on part 1

Let's create a bouncing sphere

First we need to define a material for our sphere:

```
# Start by defining a material
O.materials.append(FrictMat(young=1.0e9, poisson=0.2, density=2500, label='frictmat'))
```

`O` is our scene, it contains all the information that we need to run a DEM simulation. We can edit various components of the scene such as `materials` here. We use the python function `.append()` to add this material to the existing list of materials inside Python.

The `FrictMat` is a type of material available in Yade. Yade boasts a wide variety of materials such as `CohFrictMat`. These materials all have different constitutive laws associated with them. For now we stick with the simplest one, `FrictMat`.

Next, we need to create two spheres by appending two `bodies` to our scene, `O`:

```
O.bodies.append(
    [
    # fixed: particle's position in space will not change (support)
    sphere(center=(0, 0, 0), radius=.5, fixed=True),
    # this particles is free, subject to dynamics
    sphere((0, 0, 2), .5)
    ]
)
```

We see that we appended a `sphere` to the scene by designating its center and radius. Yade has a variety of shapes that can be appended as bodies such as `Facet`, `Box`, and others.

Now it is time to define how these spheres should move. The scene `O` has an "engines" list in `O.engines` which is a list of actions that are taken for each iteration in our simulation.

```
O.engines = [
    ForceResetter(),
    InsertionSortCollider([Bo1_Sphere_Aabb()]),
    InteractionLoop(
            [Ig2_Sphere_Sphere_ScGeom()],  # collision geometry
            [Ip2_FrictMat_FrictMat_FrictPhys()],  # collision "physics"
            [Law2_ScGeom_FrictPhys_CundallStrack()]  # contact law -- apply forces
    ),
    # Apply gravity force to particles. damping: numerical dissipation of energy.
    NewtonIntegrator(gravity=(0, 0, -9.81), damping=0.1)
]
```

Some of this may look foreign to you, but there is a logic to it. The `ForceResetter()` resets all forces stored in the scene, `InsertionSortCollider` is simply creating a sorted list of possible body interac-

tions. `InteractionLoop()` is where we assign the interaction geometry (`Ig2_Sphere_Sphere_ScGeom()`) which conveniently matches our appended body shape (`Sphere`), the collision physics `Ip2_FrictMat_FrictMat_FrictPhys()` conveniently matches our material assignment (`FrictMat`). Finally, the constitutive law is defined with `Law2_ScGeom_FrictPhys_CundallStrack()`, so we are using the classical CundallStrack contact law here. The timeintegration occurs in the last component `NewtonIntegrator` where we can add a gravitational force and damping.

Although we have already added our material, body, and engines to the scene, we should still take care to define the time step in `O.dt`:

```
O.dt = .5 * PWaveTimeStep()
```

Where the `PWaveTimeStep()` automatically estimates the critical time step associated with the stiffness of the packing. We factor that down by 1/2 to be safe, since it is but an estimate.

Starting the Script

Now that we have the full script written for our bouncing ball, it is time to start it by executing:

```
yade bouncing_sphere.py
```

in our terminal.

6.1.4 Day 1 - Yade Hands-on part 2

Building a rotating-drum

We know where to start:

```
angularVelocity = 2 #

Steel = O.materials.append(FrictMat(young=210e9, poisson=0.2, density=7200, label=
↪'Steel'))
M1 = O.materials.append(FrictMat(young=1.0e9, poisson=0.2, density=2500, label='M1'))
```

Now download the drum walls

The next step is to import the sphere and wall text files that we just downloaded:

```
from yade import ymport
facets = ymport.textFacets('Case2_Drum_Walls.txt', color=(0, 1, 0), material=Steel)
drum_ids = range(len(facets))
O.bodies.append(facets)
sp = pack.SpherePack()
sp.makeCloud(minCorner=(-0.06, -0.02, -0.06), maxCorner=(0.06, 0.02, 0.06), rMean=.
↪004, rRelFuzz=0, num=1000)
sp.toSimulation()
```

Where we are using a module called `ymport`, which has plenty of additional functionality, to import our facets. We are also introducing a very useful function called `makeCloud` which allows us to create clouds of particles with user defined properties such as the mean radius, `rMean`. Our *SpherePack()* object has a convenient method for sending the sphere pack to the simulation with `toSimulation()`.

We can now define the engine list for our rotating drum:

```
O.engines = [
    ForceResetter(),
    InsertionSortCollider([Bo1_Sphere_Aabb(), Bo1_Facet_Aabb()], label="collider"),
    InteractionLoop(
            [Ig2_Sphere_Sphere_ScGeom(), Ig2_Facet_Sphere_ScGeom()],
            [Ip2_FrictMat_FrictMat_MindlinPhys()],
```

(continues on next page)

(continued from previous page)

```
            [Law2_ScGeom_MindlinPhys_Mindlin()],
    ),
    NewtonIntegrator(damping=0, gravity=[0, 0, -9.81], label="newton"),
    RotationEngine(
            rotateAroundZero=True, zeroPoint=(0, 0, 0), rotationAxis=(0, 1, 0),␣
↪angularVelocity=angularVelocity, ids=drum_ids, label='rotation'
    ),
    VTKRecorder(iterPeriod=1000,fileName='Case2_drum-',recorders=['spheres','facets
↪'], label='vtkrecorder'),
]
```

We see some familiar commands as well as some unfamiliar ones here. `Bo1_Facet_Aabb()` and `Ig2_Facet_Sphere_ScGeom()` tell Yade that our model needs to detect collisions between our spheres and our facets. As for the contact law, this time we will use `MindlinPhys()` to determine interparticle stiffnesses for our force calculations. Again we see our familiar `NewtonIntegrator()` applying gravity to our rotating drum. But now, we add a new engine calld `RotationEngine()` which allows us to rotate bodies in our scene. We see some expected arguments to the function such as `angularVelocity` and `zeroPoint`. Finally, we want to visualize the process so we are going to add what is referred to as a `VTKRecorder()`. This will save vtk files to our disk so we can visualize them in Paraview later.

Let's now set our time-step:

```
O.dt=0.8*PWaveTimeStep()
```

Here we used our familiar `PWaveTimeStep()` to estimate the critical step. We now need to tell the scene that we are ready to start:

```
O.run()
```

Finally, we run it with our familiar command in the terminal (using the *-j* flag to indicate the number of cores we want to run the simulation on):

```
yade -j4 rotating_drum.py
```

Note: Yade will run indefinitely since we didnt provide `O.run()` with a number of iterations. The user needs to manually stop/pause when they are finished watching the simulation.

Visualizing the output files

Now that we have run our simulation and collected our vtk files using `VTKRecorder()`, we can now view those files in Paraview. Start by opening Paraview (via the GUI or via command line):

Now we can click on *file -> open* and navigate to the folder where you saved the vtk files from the rotating drum. Click on the spheres and facet files (hold ctrl to select multiple) and select *ok* from the file dialog.

Next, we will click on the green "Apply" button on the left of the window. Now we see the drum, but it is opaque, so we cant seee any particles on the inside. Paraview gives full control over the visualization of the objects. For example, we change the opacity of the drum by clicking on the *drum-facets* in the `Pipeline Browser` on the left, and then scrolling down to change the *Opacity*. Click on the green *Play* button at the top of the window to iterate thru the steps.

We see that the particles are not the proper size, so we can fix that by clicking on the *Glyph* icon right above the *Pipeline Browser* on the left. We can select the *Glyph Type*, to be *sphere* and the *Scale Array* to be *radii*. It should look something like the following image:

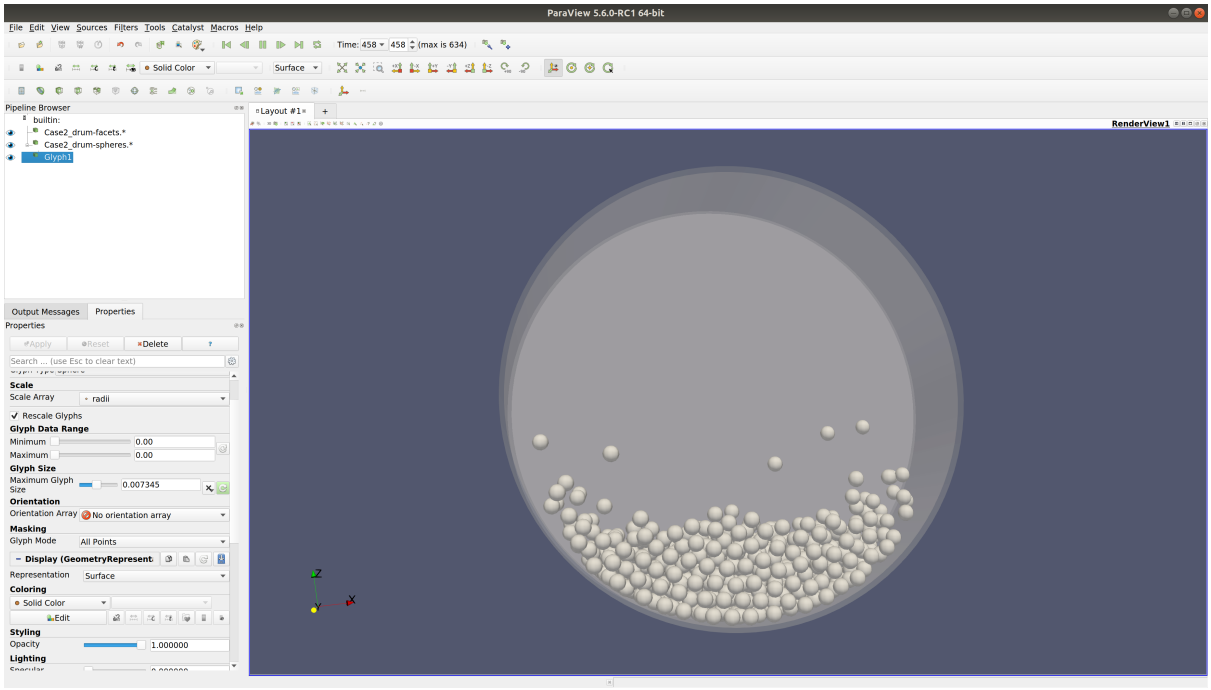


Fig. 1: Example of a paraview pipeline.

Example script

Please find a full script located in the examples folder

Today we will build a script which will simulate fluid flow through a spherical packing using Yade's FlowEngine.

6.1.5 Day 2 - Fluids Hands-on part 1

First let's import the libraries and set some parameters for future tweaking:

```
from yade import pack

num_spheres = 1000  # number of spheres
young = 1e6
compFricDegree = 3  # initial contact friction during the confining phase
finalFricDegree = 30  # contact friction during the deviatoric loading
mn, mx = Vector3(0, 0, 0), Vector3(1, 1, 1)  # corners of the initial packing
```

Next, we already know how to add materials and geometry:

```
# append sphere and wall materials
O.materials.append(FrictMat(young=young, poisson=0.5,␣
↪frictionAngle=radians(compFricDegree), density=2600, label='spheres'))
O.materials.append(FrictMat(young=young, poisson=0.5, frictionAngle=0, density=0,␣
↪label='walls'))

# create and append 4 walls of a cube sized to our mn, mx parameters
walls = aabbWalls([mn, mx], thickness=0, material='walls')
wallIds = O.bodies.append(walls)

# use makeCloud to generate a cloud of spheres inside our mn, mx bounds
sp = pack.SpherePack()
sp.makeCloud(mn, mx, -1, 0.3333, num_spheres, False, 0.95, seed=1)  #"seed" make the
```

(continues on next page)

(continued from previous page)

```
↪"random" generation always the same
sp.toSimulation(material='spheres')
```

These commands should all look familiar after passing the previous two tutorials. In brief, we are appending the *FrictMat* material type, then we assign that material to a set of walls which we then append to the scene with *O.bodies.append(walls)*. Following the walls, we create and append the spheres.

Notice how we add the walls **first** and then we add the spheres. *FlowEngine* expects by default to see the walls in the first 6 bodies (ids 0 through 5). If we need to place the walls in a different location, we can do so but we would need to set additional parameters in the *FlowEngine* engine. For now, we append the walls first.

Triaxial Stress Control

Next, we will create our *TriaxialStressController* (full parameter list with descriptions found here) and set some standard parameters to it:

```
triax = TriaxialStressController(
        internalCompaction=True,
        stressMask=7,
        goal1=-10000,
        goal2=-10000,
        goal3=-10000,
        maxMultiplier=1. + 2e4 / young,  # spheres growing factor (fast growth)
        finalMaxMultiplier=1. + 2e3 / young,  # spheres growing factor (slow growth)
)
```

Most of these parameters are geared towards how the stress is applied and achieved inside our specimen. *internalCompaction* tells *TriaxialStressController()* that we want the stress to be achieved by holding the walls fixed and growing the particles until the desired stress is achieved. *stressMask* is an integer between 0 and 7 which indicates the loading conditions (stress or strain, or which axis, more details found here). *stressMask* = 7 tells *TriaxialStressController()* that we want all axes loaded to a constant stress condition. *goalX* indicates the value along each of the three axes. So here we are asking for all 3 axes to achieve a constant compressive stress of -10000. *maxMultiplier* and *finalMaxMultiplier* control how quickly the particles can grow, more details found here.

Engine list

Next, we will set up our engine list, as usual:

```
O.engines = [
        ForceResetter(),
        InsertionSortCollider([Bo1_Sphere_Aabb(), Bo1_Box_Aabb()]),
        InteractionLoop(
                [Ig2_Sphere_Sphere_ScGeom(), Ig2_Box_Sphere_ScGeom()],
                [Ip2_FrictMat_FrictMat_FrictPhys()],
                [Law2_ScGeom_FrictPhys_CundallStrack()],
                label="iloop"
        ),
        FlowEngine(dead=1, label="flow"),  #introduced as a dead engine for the␣
↪moment, see 2nd section
        GlobalStiffnessTimeStepper(active=1, timeStepUpdateInterval=100,␣
↪timestepSafetyCoefficient=0.8),
        triax,
        NewtonIntegrator(damping=0.2, label="newton")
]
```

This should look familiar based on the previous two tutorials we completed. In summary, we need to ensure that Yade knows to expect collisions between our spheres and our walls (boxes), so we add the

Ig2_Sphere_Sphere_ScGeom() and the *Ig2_Box_sphere_ScGeom()*. Here we will stick to the classic Cundall Strack contact law. Next we add the *FlowEngine* which is set to *dead=1* so that we can run some non-flow time steps before initiating our flow simulation (see below). Here we introduce a new engine called the *GlobalStiffnessTimeStepper* which will automatically control the timestep during the simulation (see more details here). We then see the placement of our predefined *triax* followed by the familiar *NewtonIntegrator*. Our engine list now contains all the engines necessary to run a fluid-coupling simulation in Yade.

Finding an equilibrated state

But before running and fluid simulation, we need our spheres to be in a balanced and packed state. In order to achieve this, we can run some steps and check the *unbalancedForce()* while the particles grow (remember, we set *internalCompaction=True*):

```
while 1:
    O.run(1000, True)
    unb = unbalancedForce()
    if unb < 0.001 and abs(-10000 - triax.meanStress) / 10000 < 0.001:
        break
```

This while loop will start by telling Yade to run 1000 iterations through our *O.engines* list. Next it will check the total *unbalancedForce()* between all the particles. Finaly, it will ensure that the meanStress is close to our desired stress. If the unbalanced force and mean stress are not adequate, it will repeat the proces again until the break criteria is satisfied.

When this loop is completed, we know we have achieved a packed state, and we can check this visually by activating the viewer:

```
yade.qt.View()
```

It is common to keep the friction low to expedite the unbalanced force phase. But once the packing is acheived, we can simply increase the friction to match our physical properties:

```
setContactFriction(radians(finalFricDegree))
```

Setting up the *FlowEngine*

we are almost ready to run a fluid coupled test, but first we want to set up the *FlowEngine* parameters:

```
flow.dead = 0
# boundaries
flow.bndCondIsPressure = [0, 0, 1, 1, 0, 0]
flow.bndCondValue = [0, 0, 1, 0, 0, 0]
flow.boundaryUseMaxMin = [0, 0, 0, 0, 0, 0]
# permeability control
flow.permeabilityFactor = 1
flow.viscosity = 10
# remeshing criteria
```

All these parameters, and more, can be found with full descriptions here. *flow.dead = 0* tells Yade that we now want to activate the *FlowEngine*. Next we set the boundary conditions using *bndCondIsPressure* and *bndCondValue*. These tell *FlowEngine* whch boundaries should have a dirichlet boundary condition and what that pressure value should be at those boundaries. *boundaryUseMaxMin* tells *FlowEngine* if the boundary should be set automatically using max min coordinates of the bodies, or if it should use the locations of the appended walls. We appended walls and thus set all 6 components of this array to False (0).

Next we are setting the permeability parameters. *permeabilityFactor=1* tells *FlowEngine* that the permeability between pores should be set according to the Poisseuille equation. More details associated

with this parameter can be found in the class reference. Similar to *permeabilityFactor*, *viscosity* sets the viscosity used within the Poisseuille equation as well as the viscous forces.

Remeshing parameters

Understanding the remeshing methods in *FlowEngine* is integral to using the *FlowEngine* properly. During our presentations, you saw how *FlowEngine* uses a Delauay triangulation with a Voronoi dual to triangulate the pores. However, as the particles are moving, the mesh also needs to be re-computed since all the geometrical information associated with each of the pores will change (which changes permeability and force integrals). This remeshing process is expensive, so we need to find a way to remesh frequently enough that we capture the deformation, but not too frequently that the computer spends all of its time remeshing instead of running the simulation. We control the frequency of remeshing using the following parameters:

```
flow.defTolerance = 0.3
flow.meshUpdateInterval = 200
```

Where the *defTolerance* is a value which detects the maximum volumetric deformation within the system and triggers a remesh if the deformation is in excess of this value. Meanwhile, the *meshUpdateInterval* forces a remesh every XXX iterations (here we are asking for a new mesh every 200 iterations). Details about these parameters can be found here.

There are a few final settings that any *FlowEngine* user should be made aware of:

```
# solver
flow.useSolver = 3
# manually setting the timestep
O.dt = 0.1e-3
O.dynDt = False
```

Here we see a *useSolver* parameter which tells *FlowEngine* which of the various solvers we want to employ for our simulation. Both 3 and 4 are direct solvers employing a Cholesky decomposition. The difference is that 4 is more parallelized and ready for GPU acceleration. We also set the time step here manually with *O.dt* and *O.dynDt = False*. This is because there is currently no automated way to set a stable timestep for *FlowEngine* This means the user should use trial and error to find a stable timestep since it depends strongly on the dynamics/geometry of the simulation.

Getting the starting permeability

```
O.run(1, 1)
Qin = flow.getBoundaryFlux(2)
Qout = flow.getBoundaryFlux(3)
permeability = abs(Qin) / 1.e-4  #size is one, we compute K=V/H
print("Qin=", Qin, " Qout=", Qout, " permeability=", permeability)
```

We employ an easy *FlowEngine* method called *getBoundaryFlux()* for obtaining fluxes into and out of the specimen for the second and third walls in our model. We can compute the permeability here (remembering that pressure = density * gravity * head).

Starting the oedometer

The next part will require your help, we know we need new boundary conditions for the oedometer, so complete the *bndCondIsPressure* and *bndCondValue* entries below.

```
flow.bndCondIsPressure = [_, _, _, _, _, _]
flow.bndCondValue = [_, _, _, _, _, _]
flow.updateTriangulation = True  #force remeshing to reflect new BC immediately
newton.damping = 0
```

Before we start, we need to make sure we can collect data for plotting.

```
def history():
    plot.addData(
            e22=triax.strain[_],
            t=O.time,
            p=flow.getPorePressure((_, _, _)),
            s22=triax.stress(_)[_]
    )
```

We can add any data collection we wish inside this function. For example, here we will collect the triaxial strain using the *strain* function in our *TriaxialStressController*. We are also using a *FlowEngine* function called *getPorePressure* which lets use obtain the pore pressure at any user defined cooradinate. As we've mentioned before, you can find a variety of additional functions in the Yade class reference

Complete the *history()* function above before proceeding to the next code block.

We need Yade to call our *history()* function once per loop. We can do that by creating a *PyRunner*:

```
O.engines = O.engines + [PyRunner(iterPeriod=200, command='history()', label='recorder
↪')]
```

Here we are appending the *PyRunner* to our existing *O.engines* list. We are telling the *PyRunner* that we want it to call the command *history()* once every 200 ierations.

Plotting live data

Yade has a module for plotting live data, the details of the ‘plot module can be found here <https://yade-dem.org/doc/yade.plot.html>‘_tutorial-fluids

Here is an example of how we can plot the data live:

```
from yade import plot
plot.plots = {'t': (('e22', 'b--'), None, ('s22', 'g--'), ('p', 'g-'))}
plot.plot()
```

The plot module is letting us plot t vs $e22$ using a blue line (b–) for the principle y-axis. Meanwhile, it is plotting $s22$ and p using a green lines on the secondary y-axis.

We are now all set to run the fluid coupling simulation.

Example script

Please find a full script located in the examples folder

Today we will learn how to build a script that simulates heat conduction through a spherical packing and compares the numerical values to Fourier's analytical solution.

6.1.6 Day 3 - Thermal Hands-on part 1

We know where to start, let's import the necessary libraries and set our variables:

```
from yade import pack
from yade import timing
import numpy as np

num_spheres=1000
young=1e6
rad=0.003

mn,mx=Vector3(0,0,0),Vector3(1.0,0.008,0.008)
```

These are all recognizable variables from previous hands-on sessions. Next, we append our materials and walls as we've done in the past:

```
O.materials.append(FrictMat(young=young,poisson=0.5,
↪frictionAngle=radians(3),density=2600,label='spheres'))
O.materials.append(FrictMat(young=young,poisson=0.5,frictionAngle=0,density=0,label=
↪'walls'))
walls=aabbWalls([mn,mx],thickness=0,material='walls')
wallIds=O.bodies.append(walls)

O.bodies.append(pack.regularOrtho(pack.inAlignedBox(mn,mx),radius=rad,gap=-1e-8,
↪material='spheres'))
```

Here we see that we are appending a new type of sphere packing called *regularOrtho*. As the name suggests, this creates a regular orthogonal packing which will be useful for ensuring that randomness doesn't affect our comparison to the analytic conduction solution later.

Next, we need to construct our engines list as usual:

```
O.engines=[
    ForceResetter(),
    ␣
↪InsertionSortCollider([Bo1_Sphere_Aabb(aabbEnlargeFactor=intRadius),Bo1_Box_Aabb()]),
    InteractionLoop(
        ␣
↪[Ig2_Sphere_Sphere_ScGeom(interactionDetectionFactor=intRadius),Ig2_Box_Sphere_ScGeom()],
        [Ip2_FrictMat_FrictMat_FrictPhys()],
        [Law2_ScGeom_FrictPhys_CundallStrack()],label="iloop"
    ),
    FlowEngine(label="flow"),
    ThermalEngine(label='thermal'),
    GlobalStiffnessTimeStepper(active=1,timeStepUpdateInterval=100,
↪timestepSafetyCoefficient=0.8),
    VTKRecorder(iterPeriod=500,fileName='VTK'+timeStr+identifier+'/spheres-',
↪recorders=['spheres','thermal','intr'],dead=1,label='VTKrec'),
    NewtonIntegrator(damping=0.2)
]
```

Most of this should look familiar based on our previous hands-on sessions. But we see two important components including *FlowEngine* and *ThermalEngine*. These two engines rely intimately on one another for simulating THM processes, and thus *ThermalEngine cannot* be used without *FlowEngine*. We instantiate both of these engines without setting any parameters so that we can do so in detail in following steps.

We are only interested in validating the thermal conduction scheme in Yade, so we need to turn many default functionalities off starting with body dynamics:

```
for b in O.bodies:
    if isinstance(b.shape, Sphere):
            b.dynamic = False
```

b.dynamic is a body parameter which tells Yade to consider it for force calculations or not. Setting this value to false ensures that these spheres will not move during the entirety of our simulation.

Next, we set our thermal parameters:

```
thermal.conduction = True
thermal.thermoMech = False
thermal.advection = False
thermal.fluidThermoMech = False
thermal.solidThermoMech = False
thermal.fluidConduction = False
```

(continues on next page)

(continued from previous page)

```
thermal.bndCondIsTemperature = [1,1,0,0,0,0]
thermal.thermalBndCondValue = [0,0,0,0,0,0]
thermal.particleDensity = 2600 # kg/m^3
thermal.particleT0 = 400 # K
thermal.particleCp = 710 #J(kg K)
thermal.particleK = 2. #W/(mK)
thermal.particleAlpha = 11.6e-3
thermal.useKernMethod = False
```

The full set of available *ThermalEngine* parameters and all their specific details can be found here inside our Class Reference. We see that we need to ensure many of the functionalities are set to *False* for the basic conduction example here. Next, we set our boundary conditions in the same way we learned how to set boundary conditions during the previous *FlowEngine* hands-on session. Meanwhile, the initial temperature of the particles is set with *particleT0*. Finally, we set the basic thermal conduction parameters such as the particle density (*particleDensity*), thermal conductivity (*particleK*), heat capacity (*particleCp*), and diffusivity (*particleAlpha*).

Now we need to employ the *FlowEngine* for one step so that it can identify the boundaries for our *ThermalEngine*. We do not require the *FlowEngine* beyond this step because we are not simulating any fluid fluxes in the present conduction example:

```
O.dt=1.
O.dynDt=False

flow.updateTriangulation=True
flow.dead=0
flow.emulateAction()
flow.dead=1
```

Here we see that we are forcing *FlowEngine* to update the triangulation in a fake timestep with *flow.emulateAction*. Once this is done, we reset the *FlowEngine* to *dead=1* So that we do not waste computational effort calculating pressure fields.

Gathering field data

Since we are comparing our numerical conduction to an analytical scheme, we need a way to obtain field data from arbitrary coordinates. Here is an example of one way to do so:

```
def bodyByPos(x,y,z):
    cBody = O.bodies[1]
    cDist = Vector3(100,100,100)
    for b in O.bodies:
        if isinstance(b.shape, Sphere):
            dist = b.state.pos - Vector3(x,y,z)
            if np.linalg.norm(dist) < np.linalg.norm(cDist):
                cDist = dist
                cBody = b
    print('found closest body ', cBody.id, ' at ', cBody.state.pos)
    return cBody
```

Where we simply feed it arbitrary coordinates and it will return the closest body with which we can extract physical quantities such as temperature, velocity, etc.

Let's use this function to grab 10 bodies along the x-axis for us to track during the simulation:

```
axis = np.linspace(mn[0], mx[0], num=11)
axisBodies = [None] * len(axis)
```

(continues on next page)

(continued from previous page)

```
axisTrue = np.zeros(len(axis))
for i,x in enumerate(axis):
    axisBodies[i] = bodyByPos(x, mx[1]/2, mx[2]/2)
    axisTrue[i] = axisBodies[i].state.pos[0]
```

Additionally, we need a way to compute the analytical solution. Here is the solution to the heat equation for a uniform initial temperature condition and boundary conditions at 0 K:

```
k = 2
Cp = 710
rho = 2600
alpha = 6.*k/(np.pi*Cp*rho)

def analyticalHeatSolution(x,t,u0,L,alpha):
    ns = np.linspace(1,1000,1000)
    solution = 0
    for i,n in enumerate(ns):
        integral = (-2./L)*u0*L*(np.cos(n*np.pi)-1.) / (n*np.pi)
        solution += integral * np.sin(n*np.pi*x/L)*np.exp((-alpha*(n*np.pi/L)**2)*t)
    return solution
```

Where x is the x coordainte along the x-axis, t is the time of measurement, *u0* is the initial temperature of the rod, L is the length of the rod, and k is the thermal diffusivity of the rod. *alpha* is an effective thermal diffusivity which scales the discrete elements to cubical continuum elements.

Finally, we need to collect and plot the data during the simulation. The temperature can be obtained via the body state. And you have the bodies of interest set in *axisBodies*. Using the information from previous hands-on sessions, fill out the following template to collect data for

```
def history():
    plot.addData(
        t = O.time,
        i = O.iter,
        temp1 = ________,
        temp2 = ________,
        temp3 = ________,
        tempAnalytic1 = analyticalHeatSolution(________),
        tempAnalytic2 = analyticalHeatSolution(________),
        tempAnalytic3 = analyticalHeatSolution(________)
)
    plot.saveDataTxt('conductionAnalyticalComparison.txt',vars=('t','i','temp1','temp2
↪','temp3','tempAnalytic1','tempAnalytic2','tempAnalytic3'))

O.engines=O.engines+[PyRunner(iterPeriod=500,command='history()',label='recorder')]
```

Use the lessons we learned from previous hands-on sessions to:

1. plot the comparison between the numerical temperature and the analytical temperature.
2. ensure that our *VTKRecorder* is also collecting and printing files for paraview.
3. start the simulation.

Example script

Please find a full script located in the examples folder

Part 2 of our Thermal Hands-on session will focus on the full THM coupling.

6.1.7 Day 3 - Thermal Hands-on part 2

Let's build a triaxially loaded cubic specimen:

```
from yade import pack, ymport, plot, utils, export, timing
import numpy as np

young=5e6

mn,mx=Vector3(0,0,0),Vector3(0.05,0.05,0.05)

O.materials.append(FrictMat(young=young*100,poisson=0.5,frictionAngle=0,
↪density=2600e10,label='walls'))
O.materials.append(FrictMat(young=young,poisson=0.5,
↪frictionAngle=radians(30),density=2600e10,label='spheres'))

walls=aabbWalls([mn,mx],thickness=0,material='walls')
wallIds=O.bodies.append(walls)

sp=pack.SpherePack()
sp.makeCloud(mn,mx,rMean=0.0015,rRelFuzz=0.333,num=100,seed=1)
sp.toSimulation(color=(0.752, 0.752, 0.752),material='spheres')
```

Here we see that we are appending a sphere cloud to the simulation (we will compact them after setting the *O.engines* list).

Next, we need to construct our engines list as usual:

```
O.engines=[
    ForceResetter(),
    InsertionSortCollider([Bo1_Sphere_Aabb(aabbEnlargeFactor=1,label='is2aabb
↪'),Bo1_Box_Aabb()]),
    InteractionLoop(
    [Ig2_Sphere_Sphere_ScGeom(interactionDetectionFactor=1,label='ss2sc
↪'),Ig2_Box_Sphere_ScGeom()],
    [Ip2_FrictMat_FrictMat_FrictPhys()],
    [Law2_ScGeom_FrictPhys_CundallStrack()],label="iloop"
    ),
    GlobalStiffnessTimeStepper(active=1,timeStepUpdateInterval=100,
↪timestepSafetyCoefficient=0.5),
    TriaxialStressController(label='triax'),
    FlowEngine(dead=1,label="flow"),
    ThermalEngine(dead=1,label='thermal'),
    VTKRecorder(iterPeriod=500,fileName='./spheres-',recorders=['spheres','thermal',
↪'intr'],dead=1,label='VTKrec'),
    NewtonIntegrator(damping=0.5)
]
```

Now we have the full *O.engines* list set, which contains a *TriaxialStressController()* for our stress control, a *FlowEngine()* for the fluid fluxes and heat advection, and a *ThermalEngine()* for our thermal coupling.

Compacting the specimen

Let's setup the *TriaxialStressController()* for our compaction:

```
triax.maxMultiplier=1.+2e4/young
triax.finalMaxMultiplier=1.+2e3/young
triax.thickness = 0
triax.stressMask = 7
```

(continues on next page)

(continued from previous page)

```
triax.internalCompaction=True
tri_pressure = 1000
triax.goal1=triax.goal2=triax.goal3=-tri_pressure
triax.stressMask=7

while 1:
    O.run(1000, True)
    unb=unbalancedForce()
    print('unbalanced force:',unb,' mean stress: ',triax.meanStress)
    if unb<0.1 and abs(-tri_pressure-triax.meanStress)/tri_pressure<0.001:
        break

triax.internalCompaction=False
```

Here we see that we are running a loop where we run 1000 iterations of *internalCompaction* (particles grow in radius to achieve stress), then testing the *unbalancedForce()* and ultimately stopping if our stopping criteria is achieved. We can't forget that our *FlowEngine()* and *ThermalEngine()* are both set to *dead=1* in the *O.engines* list, so they will not activate during this compaction stage.

Setting up the *FlowEngine()*

```
# initial pressure condition
flow.pZero = 10
flow.meshUpdateInterval = 2
# we will activate compressibility in the fluid
flow.fluidBulkModulus = 2.2e9
flow.useSolver = 4
# enforcing a darcy permeability in the specimen
flow.permeabilityFactor = -1e-5
flow.viscosity = 0.001
# setting the boundary conditions
flow.bndCondIsPressure = [0,0,0,0,1,1]
flow.bndCondValue = [0,0,0,0,10,10]

## Thermal Stuff
flow.bndCondIsTemperature  [0,0,0,0,0,0]
# activate the thermal engine
flow.thermalEngine = True
flow.thermalBndCondValue = [0,0,0,0,0,0]
# initial temperature conditions
flow.tZero = 25

flow.dead=0
```

Setting up the *ThermalEngine()*

```
thermal.dead = 0
thermal.conduction = True
thermal.fluidConduction = True
thermal.thermoMech = True
thermal.solidThermoMech = True
thermal.fluidThermoMech = True
thermal.advection = True
thermal.useKernMethod = False
thermal.bndCondIsTemperature = [0,0,0,0,0,1]
thermal.thermalBndCondValue = [0,0,0,0,0,45]
```

(continues on next page)

(continued from previous page)

```
thermal.fluidK = 0.650
thermal.fluidBeta = 2e-5
thermal.particleT0 = 25
thermal.particleK = 2.0
thermal.particleCp = 710
thermal.particleAlpha = 3.0e-5
thermal.particleDensity = 2700
thermal.tsSafetyFactor = 0
thermal.uniformReynolds = 10
```

We won't describe each parameter here, those descriptions can be found in the Class Reference. However, it is clear we are activating conduction, advection, the thermo-fluid mechanical coupling, the solid-fluid mechanical coupling, and fluid conduction. Each component can be deactivated in case the user does not need the full THM coupling. We also see a similar assignment of boundary conditions as we saw in the previous hands-on sessions. Some additional parameters shown here include the fluid thermal conductivity (*thermal.fluidK*), the coefficient of thermal expansion for the fluid (*thermal.fluidBeta*).

Running the coupled simulation

The simulation is set and ready to run, first we will let *FlowEngine()* detect and assign the boundary conditions by running *flow.emulateAction()*:

```
O.dt=1e-4
O.dynDt=False
thermal.dead=0
flow.emulateAction()
```

Now it is up to you to finish the script

1. collect the temperature at some interesting points in the specimen
2. plot the temperature
3. export the VTK files for viewing in paraview

Example script

Please find a full script located in the examples folder

Chapter 7

Literature

7.1 Yade Technical Archive

7.1.1 About

The Yade Technical Archive (YTA) seeks to improve the reproducibility of Yade related publications by clarifying the theory that underlies Yade's opensource code, explaining algorithmic implementations, and providing practical tutorials. In doing so, YTA removes the opacity that commonly exists between readers and computational journal articles, strengthens and improves visibility of existing Yade journal papers, enables academic collaborations, and broadens open access academia.

7.1.2 Contribute

YTA seeks a variety of Yade related materials including, but not limited to:

- theoretical descriptions of code packages
- user guides and tutorials for code packages
- presentations
- course materials
- supplementary materials for journal articles

7.1.3 Contact

If you wish to contribute, please contact rob.caulk@gmail.com. Questions about individual publications are referred to the email address attached to the document description. If you have general questions regarding code, we refer you to our Q&A forum.

7.1.4 Archive

Chareyre, Bruno; Caulk, Robert; Chèvremont, William; Guntz, Thomas; Kneib, François; Kunhappen, Deepak; Pourroy, Jean (2019), Calcul distribué MPI pour la dynamique de systèmes particulaires. *Yade Technical Archive*. download full text , watch video summary , read the poster summary

Pirnia, Pouyan; Duhaime Francois; Ethier Yannic; Dubé, Jean-Sébastien (2019), COMSOL-Yade Interface (ICY) instruction guide. *Yade Technical Archive*. download full text, send an email seyedpouyan.pirnia.1@ens.etsmtl.ca , download helper files

Maurin, Raphael (2018), YADE 1D vertical VANS fluid resolution: Numerical resolution details. *Yade Technical Archive*. download full text, send an email raphael.maurin@imft.fr, follow the tutorial: *Using YADE 1D vertical VANS fluid resolution*

Maurin, Raphael (2018), YADE 1D vertical VANS fluid resolution: Theoretical basis. *Yade Technical Archive*. download full text, send an email raphael.maurin@imft.fr, follow the tutorial: *Using YADE 1D vertical VANS fluid resolution*

Maurin, Raphael (2018), YADE 1D vertical VANS fluid resolution: validations. *Yade Technical Archive*. download full text, send an email raphael.maurin@imft.fr, follow the tutorial: *Using YADE 1D vertical VANS fluid resolution*

Caulk, Robert (2018), Stochastic Augmentation of the Discrete Element Method for Investigation of Tensile Rupture in Heterogeneous Rock. *Yade Technical Archive*. DOI 10.5281/zenodo.1202039. download full text , send an email rob.caulk@gmail.com , follow the tutorial: *Simulating Acoustic Emissions in Yade*

7.2 Yade publications

Publications on Yade itself, or produced with Yade, are listed on this page.

The first section gives the reference that we kindly ask you to use for citing Yade in publications, as explained in the "Acknowledging Yade" section.

With the increasing rate of publications using Yade, it became difficult to list them all; therefore, coverage of recent years is only partial.

You can help us: if you publish (or know) a publication related to Yade, please add it to this list.

- Preferred: open a merge request on GitLab:

  https://gitlab.com/yade-dev/trunk/-/merge_requests

If a PDF is freely available, please include a URL for direct full-text download. Yade's web server can host such PDFs if legally permitted.

Note

This file is generated from doc/yade-articles.bib, doc/yade-conferences.bib, doc/yade-theses.bib, and doc/yade-tech-archive.bib.

7.2.1 Journal articles

7.2.2 Conference materials and book chapters

7.2.3 Master and PhD theses

7.2.4 *Yade Technical Archive*

7.3 References

All external articles referenced in Yade documentation.

Note

This file is generated from doc/references.bib.

Chapter 8

Yade community events

8.1 Yade community events

8.1.1 2nd Yade hackathon

The second Yade Hackathon took place in Amsterdam, Netherlands on the 16th and 17th of July, 2026. Thanks to the local organization of Robert Caulk and to the financial support of the Aqua division of INRAE French research institute, up to 9 developers joined on site and remotely to discuss and code YADE features.

Aligned with modern trends, a significant share of the discussions devoted to the possible use of Artificial Intelligence (AI) for helping users to define YADE simulations scripts. Amsterdam temperatures were actually cool enough on those days such that global warming was out of the picture anyway.

Physics-based discussions also took place concerning the various YADE implementations of contact models with a visco-elastic component; as well as about the numerical (so-called "Cundall") damping, which is now absent from any YADE simulation by default.

Once all corresponding gitlab.com "merge request" (MR) processes will complete (e.g., MR 1190, MR 1194, MR 1195) and thanks also to an improved on-line documentation (MR 1196), we hope this will eventually benefit every YADE user !

8.1.2 1st Yade hackathon

The first Yade Hackathon took place in Freiberg, Germany on June 23rd and 24th of 2022!

Eight developers were on site while one joined remotely during the presentation sessions. This tight knit group of Yade developers already meets bi-monthly via zoom, but the Yade Hackathon gave them the opportunity to meet offline to discuss issues in the software, prognose the future of the project, fix bugs, work on the code, and even tour the city together.

A former core developer, Vaclav Smilauer, made a special guest appearance during the first day where he even contributed some expert advice to the hackathon.

Main topics of discussion included:

- Install documentation update, e.g., split installation dependencies, depending on the required features. MR
- Drop the wiki page and moving the valuable information to the documentation or on the website. Issue
- Check the last publications, where the Yade was cited and put the links into the documentation. MR
- Add support of the Qt6, which is already available in Debian repositories Issue. Follow up actions are identified.

Fig. 1: From left to right: Adarsh Dwivedi, Bruno Chareyre, Katia Boschi, Luc Scholtès, Oussama Larkem, Jérôme Duriez, Robert Caulk (not pictured Vasileios Angelidakis and Karol Brzeziński)

Fig. 2: From left to right, Vasileios Angelidakis, Anton Gladky, Katia Boschi, Jérôme Duriez, Robert Caulk, Bruno Chareyre, Janek Kozicki, Vaclav Smilauer (not pictured Klaus Theoni)

- Distribute the yade-dem.org domain permissions to increase the bus factor for the project.
- Fix the newly added gitlab runner nova1. Issue
- Present the latest work, based on Yade and newer features added in the source code recently.
- Drop google-analytics code from the website. Issue, MR.
- Create a Yade short-course section on the website and include all content associated with a recent short-course. MR
- Discuss a future paper in Computer Physics Communications.
- Explore technical and non-technical discussions.

In contrast to their traditional means of communication via week-long email and issue tracking exchanges, the Yade developers found that this offline Hackathon provided an opportunity to quickly discuss problems and solutions.

We want to thank everybody who made this event possible:

- TU Bergakademie Freiberg for a general support.
- Institute for Informatics of the TU Bergakademie Freiberg, and personally Christian Schubert, Birgit Steffen and Sebastian Zug.
- Institute for the processing machines and recycling system technick, TU Bergakademie Freiberg, and personally Dr.-Ing. Prof. Holger Lieberwirth
- Institute of dynamics and flow mechanics, TU Bergakademie Freiberg, and personally Dr.-Ing. Prof. Rüdiger Schwarze
- Haver Engineering GmbH, and personally Jan Lampke and Hagen Müller.

Improvement plans for the next Hackathon include:

- Block out more time for the hacking. Ideally - many more days and until the evening
- Locate a better meeting place for the international group (Frankfurt-Am-Main)

8.1.3 2nd Yet Another Discrete Element Workshop

Aix-en-Provence, April 26-27, 2018

Web-Site about the workshop

8.1.4 1st Yet Another Discrete Element Workshop

Grenoble, July 7-9, 2014

NEW: The booklet of presentations is available

Objectives

The **1st Yade Workshop** will be held on **7-9 July in Grenoble, France**.

Particle-scale modeling remains an area of active developments, decades after the pioneering work of P. Cundall. A large part of these developments is mirrored by contributions to the open source platform Yade-DEM.

The objective of this workshop is to gather people interested in DEM and DEM-related developments, with special focus on new models and couplings, algorithmic issues, performance and parallelization.

Download the program

Check also the access map

The list of sessions includes:

- DEM applications

- Numerical methods and modeling techniques
- Complex shapes
- Multiphase couplings

The last day will take the form of short talks and less formal meetings focused on the Yade-DEM project. A coding and brainstorming session will be organised for Yade devs, and coordinated by **Klaus Thoeni**. For more info see Brainstorming.

The **1st Yade Workshop** will be an opportunity for the people developing Yade-DEM and other DEM codes to meet each other, share ideas and elaborate workplans and cooperations.

Contact & Registration

We will still accept a limited number of registrations

For registration and practical informations email us.

Invited Speakers

- **Anton Gladky**(TU Freiberg)
- **Klaus Thoeni** (Univ. Newcastle AU)
- **Alexander Eulitz** (TU Berlin)
- **Frédéric Donzé**(Univ. Grenoble)
- **Jan Stránský** (TU Prague)
- **Jérôme Duriez**(Univ. Grenoble)
- **Christian Jakob** (TU Freiberg)
- **Burak ER**
- **François Kneib**(IRSTEA Grenoble)
- **Václav Šmilauer** (Prague)
- **Ricardo Pieralisi** (Univ. Catalunya)
- **Janek Kozicki** (TU Gdansk)
- **Emanuele Catalano** (Itasca CG Lyon)

Organizing Comitee

- Bruno Chareyre (Grenoble Inst. of Tech. / 3SR Lab)
- Caroline Chalak (Grenoble Inst. of Tech. / 3SR Lab)

Acknowledgement

The workshop is supported by

- Maimosine
- Fédération VOR
- Fédération 3G
- Université Grenoble Alpes
- région Rhône-Alpes.

Chapter 9

Indices and tables

- genindex
- modindex
- search

Bibliography

[Abdallah2022] Abdallah, Ali, Aboul Hosn, Rodaina, Al Tfaily, Bilal, Sibille, Luc (2022), **Identifying parameters of a discrete numerical model of soil from a geotechnical field test**. *European Journal of Environmental and Civil Engineering*, pages 1–20.

[Abdallah2024] Abdallah, Ali, Vincens, Eric, Magoariec, Hél'ene, Picault, Christophe (2024), **DEM filtration modelling for granular materials: Comparative analysis of dry and wet approaches**. *International Journal for Numerical and Analytical Methods in Geomechanics* (48), pages 870–886.

[Abdi2022] Abdi, Rezvan, Krzaczek, Marek, Tejchman, J (2022), **Comparative study of high-pressure fluid flow in densely packed granules using a 3D CFD model in a continuous medium and a simplified 2D DEM-CFD approach**. *Granular Matter* (24), pages 1–25.

[Abdi2023] Abdi, Rezvan, Krzaczek, Marek, Tejchman, J (2023), **Simulations of high-pressure fluid flow in a pre-cracked rock specimen composed of densely packed bonded spheres using a 3D CFD model and simplified 2D coupled CFD-DEM approach**. *Powder Technology* (417), pages 118238.

[Aboul2017] Aboul Hosn, R., Sibille, L., Benahmed, N., Chareyre, B. (2017), **Discrete numerical modeling of loose soil with spherical particles and interparticle rolling friction**. *Granular Matter* (19). DOI 10.1007/s10035-016-0687-0

[Aboul2018] Aboul R. Hosn, L. Sibille, N. Benahmed, B. Chareyre (2018), **A discrete numerical model involving partial fluid-solid coupling to describe suffusion effects in soils**. *Computers and Geotechnics* (95), pages 30–39. DOI https://doi.org/10.1016/j.compgeo.2017.11.006

[Albaba2015] Albaba, A, Lambert, S, Nicot, F, Chareyre, B (2015), **Relation between microstructure and loading applied by a granular flow to a rigid wall using DEM modeling**. *Granular Matter* (17), pages 603–616. DOI 10.1007/s10035-015-0579-8

[Ali2024] Ali, Usman, Kikumoto, Mamoru, Ciantia, Matteo Oryem, Cui, Ying, Previtali, Marco (2024), **Why Modeling Particle Shape Matters: Significance of Particle-Scale Modeling in Describing Global and Local Granular Responses**. *Journal of Geotechnical and Geoenvironmental Engineering* (150), pages 04024079.

[AlTfaily2024] Al Tfaily, Bilal, Sibille, Luc, Hosn, Rodaina Aboul, Bennabi, Abdelkrim (2024), **Prediction ability of discrete element model of loose granular media subjected to complex loadings**. *Powder Technology* (433), pages 119251.

[Angelidakis2021] Angelidakis, Vasileios, Nadimi, Sadegh, Utili, Stefano (2021), **SHape Analyser for Particle Engineering (SHAPE): Seamless characterisation and simplification of particle morphology from imaging data**. *Computer Physics Communications*, pages 107983.

[Angelidakis2024] Angelidakis, Vasileios, Boschi, Katia, Brzezinski, Karol, Caulk, Robert A, Chareyre, Bruno, Del Valle, Carlos Andrés, Duriez, Jérôme, Gladky, Anton, Van Der Haven, Dingeman LH, Kozicki, Janek, others (2024), **YADE-An extensible framework for the interactive simulation of multiscale, multiphase, and multiphysics particulate systems**. *Computer Physics Communications* (304), pages 109293.

[Asadi2022] Asadi, Mohsen, Mahboubi, Ahmad, Thoeni, Klaus (2022), **Towards more realistic modelling of sand-rubber mixtures considering shape, deformability and micromechanics**. *Canadian Geotechnical Journal*.

[Audry2023] Audry, Nils, Harthong, Barthélémy, Imbault, Didier (2023), **Comparison between periodic and non-periodic boundary conditions in the multi-particle finite element modelling of ductile powders**. *Powder Technology* (429), pages 118871.

[Audry2024] Audry, Nils, Harthong, Barthélémy, Imbault, Didier (2024), **The mesoscale mechanics of compacted ductile powders under shear and tensile loads**. *Journal of the Mechanics and Physics of Solids*, pages 105807.

[Bance2014] Bance, S., Fischbacher, J., Schrefl, T., Zins, I., Rieger, G., Cassignol, C. (2014), **Micromagnetics of shape anisotropy based permanent magnets**. *Journal of Magnetism and Magnetic Materials* (363), pages 121–124.

[Barbosa2022b] Barbosa, Luis Alfredo Pires, Gerke, Kirill M, Gerke, Horst H (2022), **Modelling of soil mechanical stability and hydraulic permeability of the interface between coated biopore and matrix pore regions**. *Geoderma* (410), pages 115673.

[Barbosa2022a] Barbosa, Luis Alfredo Pires, Gerke, Kirill M, Munkholm, Lars J, Keller, Thomas, Gerke, Horst H (2022), **Discrete element modeling of aggregate shape and internal structure effects on Weibull distribution of tensile strength**. *Soil and Tillage Research* (219), pages 105341.

[Barbosa2020b] Barbosa, Luis Alfredo Pires, Keller, Thomas, de Oliveira Ferraz, Antonio Carlos (2020), **Scale effect of aggregate rupture: Using the relationship between friability and fractal dimension to parameterise discrete element models**. *Powder Technology* (375), pages 327–336.

[Barbosa2020] Barbosa, Luis Alfredo Pires (2020), **Modelling the aggregate structure of a bulk soil to quantify fragmentation properties and energy demand of soil tillage tools in the formation of seedbeds**. *Biosystems Engineering* (197), pages 203–215.

[Barros2023] Barros, Guilherme, Pereira, Andre, Rojek, Jerzy, Carter, John, Thoeni, Klaus (2023), **Efficient multi-scale staggered coupling of discrete and boundary element methods for dynamic problems**. *Computer Methods in Applied Mechanics and Engineering* (415), pages 116227.

[Basson2023] Basson, Mandeep Singh, Martinez, Alejandro (2023), **Numerical and experimental estimation of anisotropy in granular soils using multi-orientation shear wave velocity measurements**. *Granular Matter* (25), pages 55.

[Basson2024a] Basson, Mandeep Singh, Martinez, Alejandro, DeJong, Jason T (2024), **DEM investigation of the effect of gradation on the strength, dilatancy, and fabric evolution of coarse-grained soils**. *Journal of Geotechnical and Geoenvironmental Engineering* (150), pages 04024060.

[Basson2024b] Basson, Mandeep Singh, Martinez, Alejandro, DeJong, Jason T (2024), **DEM simulations of the liquefaction resistance and post-liquefaction strain accumulation of coarse-grained soils with varying gradations**. *Computers and Geotechnics* (174), pages 106649.

[Benniou2020] Benniou, H., Accary, A., Malecot, Y., Briffaut, M., Daudeville, L. (2020), **Discrete element modeling of concrete under high stress level: influence of saturation ratio**. *Computational Particle Mechanics*. DOI 10.1007/s40571-020-00318-5

[Binelo2022] Binelo, Manuel O, Lima, Rodolfo F, Faoro, Vanessa, Binelo, Marcia FB (2022), **Computational modelling of a grain spreader for use in silos**. *Biosystems Engineering* (223), pages 29–40.

[Bonilla2015] Bonilla-Sierra, V., Scholtès, L., Donzé, F.V., Elmouttie, M.K. (2015), **Rock slope stability analysis using photogrammetric data and DFN–DEM modelling**. *Acta Geotechnica*, pages 1–15. DOI 10.1007/s11440-015-0374-z

[Boon2015] Boon, C.W., Houlsby, G.T., Utili, S. (2015), **A new rock slicing method based on linear programming**. *Computers and Geotechnics* (65), pages 12–29. DOI 10.1016/j.compgeo.2014.11.007

[Boon2015b] Boon, C.W., Houlsby, G.T., Utili, S. (2015), **Designing Tunnel Support in Jointed Rock Masses Via the DEM**. *Rock Mechanics and Rock Engineering* (48), pages 603–632. DOI 10.1007/s00603-014-0579-8

[Boon2014] Boon, C.W., Houlsby, G.T., Utili, S. (2014), **New insights into the 1963 Vajont slide using 2D and 3D distinct-element method analyses**. *Géotechnique* (64), pages 800–816. DOI 10.1680/geot.14.P.041

[Boon2013] Boon, C.W., Houlsby, G.T., Utili, S. (2013), **A new contact detection algorithm for three-dimensional non-spherical particles**. *Powder Technology* (248), pages 94–102. DOI 10.1016/j.powtec.2012.12.040

[Boon2012] Boon, C.W., Houlsby, G.T., Utili, S. (2012), **A new algorithm for contact detection between convex polygonal and polyhedral particles in the discrete element method**. *Computers and Geotechnics* (44), pages 73–82. DOI 10.1016/j.compgeo.2012.03.012

[Boschi2025] Boschi, Katia, Chareyre, Bruno, di Prisco, Claudio (2025), **DEM-PFV numerical investigation of the spatial heterogeneity induced by suffusion in sands**. *Computers and Geotechnics* (188), pages 107525.

[Bourrier2013] Bourrier, F., Kneib, F., Chareyre, B., Fourcaud, T. (2013), **Discrete modeling of granular soils reinforcement by plant roots**. *Ecological Engineering*. DOI 10.1016/j.ecoleng.2013.05.002

[Bourrier2015] Bourrier, F., Lambert, S., Baroth, J. (2015), **A reliability-based approach for the design of rockfall protection fences**. *Rock Mechanics and Rock Engineering* (48), pages 247–259.

[Brzezinski2022] Brzezinski Karol, Anton Gladky (2022), **Clump breakage algorithm for DEM simulation of crushable aggregates**. *Tribology International* (173), pages 107661. DOI https://doi.org/10.1016/j.triboint.2022.107661

[Brzezinski2023a] Brzezinski Karol, Zbiciak Artur, Anton Gladky (2023), **Implementation of a viscoelastic boundary condition to Yade – open source DEM software**. *Journal of Theoretical and Applied Mechanics* (62), pages 355–364. DOI https://doi.org/10.15632/jtam-pl/163053

[Brzezinski2023b] Brzezinski, Karol, Ciezkowski, Pawel, Bak, Sebastian (2023), **Tricking the fractal nature of granular materials subjected to crushing**. *Powder Technology* (425), pages 118601.

[Brzezinski2025] Brzezinski, Karol (2025), **Evaluating the plate compactor frequency effect on compaction efficiency: numerical study with discrete element method**. *Granular Matter* (27), pages 35.

[Canbolat2025] Canbolat, Ahmet Utku, Nadimi, Sadegh, Angelidakis, Vasileios (2025), **A Python implementation of CLUMP, the Code Library to generate Universal Multi-sphere Particles**. *SoftwareX* (29), pages 101957.

[Catalano2014a] Catalano, E., Chareyre, B., Barthélémy, E. (2014), **Pore-scale modeling of fluid-particles interaction and emerging poromechanical effects**. *International Journal for Numerical and Analytical Methods in Geomechanics* (38), pages 51–71. DOI 10.1002/nag.2198 (http://arxiv.org/pdf/1304.4895.pdf)

[Caulk2019] Caulk Robert A., Emanuele Catalano, Bruno Chareyre (2019), **Accelerating Yade's poromechanical coupling with matrix factorization reuse, parallel task management, and GPU computing**. *Computer Physics Communications*, pages 106991. DOI https://doi.org/10.1016/j.cpc.2019.106991

[Caulk2020b] Caulk Robert, Luc Sholtès, Marek Krzaczek, Bruno Chareyre (2020), **A pore-scale thermo–hydro-mechanical model for particulate systems**. *Com-*

puter Methods in Applied Mechanics and Engineering (372), pages 113292. DOI https://doi.org/10.1016/j.cma.2020.113292

[Caulk2020] Caulk, Robert A. (2020), **Modeling acoustic emissions in heterogeneous rocks during tensile fracture with the Discrete Element Method**. *Open Geomechanics* (2). DOI 10.5802/ogeo.5

[Chalak2017] Chalak, C., Chareyre, B., Nikooee, E., Darve, F. (2017), **Partially saturated media: from DEM simulation to thermodynamic interpretation**. *European Journal of Environmental and Civil Engineering* (21), pages 798–820. DOI 10.1080/19648189.2016.1164087

[Chapelle2021] Chapelle, David, Maynadier, Anne, Bebon, Ludovic, Thiébaud, Frédéric (2021), **Hydrogen Storage: Different Technologies, Challenges and Stakes. Focus on TiFe Hydrides**. In *Advances in Renewable Hydrogen and Other Sustainable Energy Carriers* Springer ,

[Chareyre2012a] Chareyre, B., Cortis, A., Catalano, E., Barthélemy, E. (2012), **Pore-Scale Modeling of Viscous Flow and Induced Forces in Dense Sphere Packings**. *Transport in Porous Media* (92), pages 473–493. DOI 10.1007/s11242-011-9915-6

[Chassagne2020] Chassagne, Rémi, Frey, Philippe, Maurin, Raphaël, Chauchat, Julien (2020), **Mobility of bidisperse mixtures during bedload transport**. *Physical Review Fluids* (5), pages 114307.

[Chassagne2023] Chassagne, Rémi, Bonamy, Cyrille, Chauchat, Julien (2023), **A frictional–collisional model for bedload transport based on kinetic theory of granular flows: discrete and continuum approaches**. *Journal of Fluid Mechanics* (964), pages A27.

[Chen2007] Chen, F., Drumm, E. C., Guiochon, G. (2007), **Prediction/Verification of Particle Motion in One Dimension with the Discrete-Element Method**. *International Journal of Geomechanics, ASCE* (7), pages 344–352. DOI 10.1061/(ASCE)1532-3641(2007)7:5(344)

[Chen2011a] Chen, F., Drumm, E., Guiochon G. (2011), **Coupled discrete element and finite volume solution of two classical soil mechanics problems**. *Computers and Geotechnics*. DOI 10.1016/j.compgeo.2011.03.009

[Chen2014] Chen, J., Huang, B., Shu, X., Hu, C. (2014), **DEM Simulation of Laboratory Compaction of Asphalt Mixtures Using an Open Source Code**. *Journal of Materials in Civil Engineering*.

[Chen2012] Chen, Jingsong, Huang, Baoshan, Chen, Feng, Shu, Xiang (2012), **Application of discrete element method to Superpave gyratory compaction**. *Road Materials and Pavement Design* (13), pages 480–500. DOI 10.1080/14680629.2012.694160

[Cheng2016] Cheng, H., Yamamoto, H., Thoeni, K. (2016), **Numerical study on stress states and fabric anisotropies in soilbags using the DEM**. *Computers and Geotechnics* (76), pages 170–183. DOI 10.1016/j.compgeo.2016.03.006

[Chevremont2020] Chèvremont, William, Bodiguel, Hugues, Chareyre, Bruno (2020), **Lubricated contact model for numerical simulations of suspensions**. *Powder Technology* (372), pages 600–610.

[Chevremont2019] Chèvremont, William, Chareyre, Bruno, Bodiguel, Hugues (2019), **Quantitative study of the rheology of frictional suspensions: Influence of friction coefficient in a large range of viscous numbers**. *Physical Review Fluids* (4), pages 064302.

[Chodkiewicz2025] Chodkiewicz, Pawel, Zalewski, Robert, Lengiewicz, Jakub (2025), **A fully discrete element approach for modeling vacuum packed particle dampers**. *International Journal of Mechanical Sciences* (22), pages 110922.

[Coulibaly2020] Coulibaly, Jibril B, Shah, Manan, Loria, Alessandro F Rotta (2020), **Thermal cycling effects on the structure and physical properties of granular materials**. *Granular Matter* (22), pages 1–19.

[Cuomo2016] Cuomo, S., Chareyre, B., d'Arista, P., Sala, M.D., Cascini, L. (2016), **Micromechanical modelling of rainsplash erosion in unsaturated soils by Discrete Element Method**. *CATENA* (147), pages 146–152.

[Dang2010a] Dang, H. K., Meguid, M. A. (2010), **Algorithm to Generate a Discrete Element Specimen with Predefined Properties**. *International Journal of Geomechanics* (10), pages 85–91. DOI 10.1061/(ASCE)GM.1943-5622.0000028

[Dang2010b] Dang, H. K., Meguid, M. A. (2010), **Evaluating the performance of an explicit dynamic relaxation technique in analyzing non-linear geotechnical engineering problems**. *Computers and Geotechnics* (37), pages 125–131. DOI 10.1016/j.compgeo.2009.08.004

[delValle2024] del Valle, Carlos Andrés, Angelidakis, Vasileios, Roy, Sudeshna, Mu~noz, José Daniel, P"oschel, Thorsten (2024), **SPIRAL: An efficient algorithm for the integration of the equation of rotational motion**. *Computer Physics Communications* (297), pages 109077.

[DePue2019] De Pue Jan, Gemmina Di Emidio, R. Daniel Verastegui Flores, Adam Bezuijen, Wim M. Cornelis (2019), **Calibration of DEM material parameters to simulate stress-strain behaviour of unsaturated soils during uniaxial compression**. *Soil and Tillage Research* (194), pages 104303. DOI https://doi.org/10.1016/j.still.2019.104303

[DePue2019b] De Pue, Jan, Cornelis, Wim M (2019), **DEM simulation of stress transmission under agricultural traffic Part 1: Comparison with continuum model and parametric study**. *Soil and Tillage Research* (195), pages 104408. DOI https://doi.org/10.1016/j.still.2019.104408

[DePue2020a] De Pue, Jan, Lamandé, Mathieu, Cornelis, Wim M (2020), **DEM simulation of stress transmission under agricultural traffic Part 2: Shear stress at the tyre-soil interface**. *Soil and Tillage Research* (203), pages 104660. DOI https://doi.org/10.1016/j.still.2020.104660

[DePue2020b] De Pue, Jan, Lamandé, Mathieu, Schjonning, Per, Cornelis, Wim M (2020), **DEM simulation of stress transmission under agricultural traffic Part 3: Evaluation with field experiment**. *Soil and Tillage Research* (200), pages 104606. DOI https://doi.org/10.1016/j.still.2020.104606

[DeSimone2025] De Simone, Marcelo, Souza, Lourdes MS, Roehl, Deane (2025), **A DEM Model for Assessing the Mechanical Effects of CO2 Alteration in a Carbonate Rock**. *International Journal for Numerical and Analytical Methods in Geomechanics*.

[Deng2021] Deng, Na, Wautier, Antoine, Thiery, Yannick, Yin, Zhen-Yu, Hicher, Pierre-Yves, Nicot, Franccois (2021), **On the attraction power of critical state in granular materials**. *Journal of the Mechanics and Physics of Solids* (149), pages 104300.

[Dincc2023] Dincc G"oug"ucs, "Ozge, Avcsar, Elif, Develi, Kayhan, cCalik, Ayten (2023), **Quantifying the Rock Damage Intensity Controlled by Mineral Compositions: Insights from Fractal Analyses**. *Fractal and Fractional* (7), pages 383.

[Donze2008] Donzé, F.V. (2008), **Impacts on cohesive frictional geomaterials**. *European Journal of Environmental and Civil Engineering* (12), pages 967–985.

[Donze2021] Donzé, Frédéric-Victor, Klinger, Yann, Bonilla-Sierra, Viviana, Duriez, Jérôme, Jiao, Liqing, Scholtès, Luc (2021), **Assessing the brittle crust thickness from strike-slip fault segments on Earth, Mars and Icy moons**. *Tectonophysics* (805), pages 228779.

[Duriez2011] Duriez J., Darve F., Donzé F.V. (2011), **A discrete modeling-based constitutive relation for infilled rock joints**. *International Journal of Rock Mechanics & Mining Sciences* (48), pages 458–468. DOI 10.1016/j.ijrmms.2010.09.008

[Duriez2013] Duriez J., Darve F., Donzé F.V. (2013), **Incrementally non-linear plasticity applied to rock joint modelling**. *International Journal for Numerical and Analytical Methods in Geomechanics* (37), pages 453–477. DOI 10.1002/nag.1105

[Duriez2016] Duriez J., Scholtès L., Donzé F.V. (2016), **Micromechanics of wing crack propagation for different flaw properties**. *Engineering Fracture Mechanics* (153), pages 378 – 398. DOI 10.1016/j.engfracmech.2015.12.034

[Duriez2016b] Duriez J., R. Wan (2016), **Stress in wet granular media with interfaces via homogenization and discrete element approaches**. *Journal of Engineering Mechanics* (142). DOI 10.1061/(ASCE)EM.1943-7889.0001163

[Duriez2017] Duriez J., R. Wan (2017), **Subtleties in discrete-element modelling of wet granular soils**. *Géotechnique* (67), pages 365–370. DOI 10.1680/jgeot.15.P.113

[Duriez2017b] Duriez J., R. Wan (2017), **Contact angle mechanical influence for wet granular soils**. *Acta Geotechnica* (12), pages 67–83. DOI 10.1007/s11440-016-0500-6

[Duriez2017c] Duriez, J., Eghbalian, M., Wan, R., Darve, F. (2017), **The micromechanical nature of stresses in triphasic granular media with interfaces**. *Journal of the Mechanics and Physics of Solids* (99), pages 495–511. DOI 10.1016/j.jmps.2016.10.011

[Duriez2018] Duriez J., R. Wan, M. Pouragha Mehdi, F. Darve **Revisiting the existence of an effective stress for wet granular soils with micromechanics**. *International Journal for Numerical and Analytical Methods in Geomechanics* (42), pages 959–978. DOI 10.1002/nag.2774

[Duriez2018b] Duriez J., R. Wan (2018), **A micromechanical UNSAT effective stress expression for stress-strain behaviour of wet granular materials**. *Geomechanics for Energy and the Environment* (15), pages 10–18. DOI 10.1016/j.gete.2017.12.003

[Duriez2021a] Duriez J., S. Bonelli (2021), **Precision and computational costs of Level Set-Discrete Element Method (LS-DEM) with respect to DEM**. *Computers and Geotechnics* (134), pages 104033. DOI 10.1016/j.compgeo.2021.104033

[Duriez2021b] Duriez, Jérôme, Galusinski, Cédric (2021), **A Level Set-Discrete Element Method in YADE for numerical, micro-scale, geomechanics with refined grain shapes**. *Computers and Geosciences* (157), pages 104936. DOI 10.1016/j.cageo.2021.104936

[Duriez2025] Duriez, Jérôme, Galusinski, Cédric (2025), **Faster and objective Level Set-DEM mechanical simulations of discrete systems with convex particles from contact history and particle surface considerations**. *Computer Physics Communications* (316), pages 109803. DOI 10.1016/j.cpc.2025.109803

[Galarraga2026] Galárraga, M., Scholtès, L., Chevalier, B., Aisyah, N., Kelfoun, K. (2026), **Insights Into the 2020 Instability Crisis of Merapi Through Numerical Modeling**. *Journal of Geophysical Research: Solid Earth* (131), pages e2025JB032728. DOI https://doi.org/10.1029/2025JB032728

[Dyck2015] Dyck, N.J., Straatman, A.G. (2015), **A new approach to digital generation of spherical void phase porous media microstructures**. *International Journal of Heat and Mass Transfer* (81), pages 470–477.

[Effeindzourou2016] Effeindzourou, A., Chareyre, B., Thoeni, K., Giacomini, A., Kneib, F. (2016), **Modelling of deformable structures in the general framework of the discrete element method**. *Geotextiles and Geomembranes* (44), pages 143–156. DOI 10.1016/j.geotexmem.2015.07.015

[Elias2014] Eliáš Jan (2014), **Simulation of railway ballast using crushable polyhedral particles**. *Powder Technology* (264), pages 458–465. DOI 10.1016/j.powtec.2014.05.052

[Epifancev2013] Epifancev, K., Nikulin, A., Kovshov, S., Mozer, S., Brigadnov, I. (2013), **Modeling of Peat Mass Process Formation Based on 3D Analysis of the Screw Machine by the Code YADE**. *American Journal of Mechanical Engineering* (1), pages 73–75. DOI 10.12691/ajme-1-3-3

[Epifantsev2012] Epifantsev, K., Mikhailov, A., Gladky, A. (2012), **Proizvodstvo kuskovogo torfa, ekstrudirovanie, forma zakhodnoi i kalibriruyushchei chasti fil'ery matritsy, metod diskretnykh elementov [RUS]**. *Mining informational and analytical bulletin (scientific and technical journal)*, pages 212–219.

[Escobar2023] Escobar, Andrés, Guillard, Franccois, Einav, Itai, Faug, Thierry (2023), **A scaling law for the length of granular jumps down smooth inclines**. *Journal of Fluid Mechanics* (973), pages R1.

[Escobar2025] Escobar, Andres, Baker, James, Guillard, Franccois, Faug, Thierry, Einav, Itai (2025), **Experimental confirmation of secondary flows within granular media**. *Nature Communications* (16), pages 7446.

[Farahnak2024a] Farahnak, Mojtaba, Wan, Richard, Pouragha, Mehdi, Nicot, Franccois (2024), **A multiscale bifurcation analysis using micromechanical-based constitutive tensor for granular material**. *International Journal of Solids and Structures* (298), pages 112866.

[Farahnak2024b] Farahnak, Mojtaba, Wan, Richard, Pouragha, Mehdi (2024), **Exploring the tensorial nature of capillary stress and the constitutive role of contact stress in wet granular materials**. *Computers and Geotechnics* (173), pages 106492.

[Fathipour2025] Fathipour-Azar, Hadi, Duriez, Jérôme (2025), **A Comparative Study of Existing and New Sphere Clump Generation Algorithms for Modeling Arbitrary Shaped Particles**. *Archives of Computational Methods in Engineering* (32), pages 4033–4048. DOI 10.1007/s11831-025-10256-1

[Faulconnier2025] Faulconnier, Antoine, Job, Stéphane, Brocail, Julien, Peyret, Nicolas, Dion, Jean-Luc (2025), **Elasto-frictional reduced model of a cyclically sheared container filled with particles**. *Granular Matter* (27), pages 104.

[Favier2009a] Favier, L., Daudon, D., Donzé, F.V., Mazars, J. (2009), **Predicting the drag coefficient of a granular flow using the discrete element method**. *Journal of Statistical Mechanics: Theory and Experiment* (2009), pages P06012.

[Favier2012] Favier, L., Daudon, D., Donzé, F.V. (2012), **Rigid obstacle impacted by a supercritical cohesive granular flow using a 3D discrete element model**. *Cold Regions Science and Technology* (85), pages 232–241.

[Feng2021] Feng, Shaochuan, Kamat, Amar M, Sabooni, Soheil, Pei, Yutao (2021), **Experimental and numerical investigation of the origin of surface roughness in laser powder bed fused overhang regions**. *Virtual and Physical Prototyping*, pages 1–19.

[Feng2025] Feng, Shi-Jin, Wang, Ya-Qiong (2025), **3D discrete-element analysis of arching and anchoring within the geogrid-reinforced shallow embankment overlying a void**. *Computers and Geotechnics* (186), pages 107440.

[Firouzabadi2023] Firouzabadi, Mahdeyeh, Esmaeili, Kamran, Rashkolia, Gholamreza Saeedi, Asadi, Mohsen (2023), **A discrete element modelling of gravity flow in sublevel caving considering the shape and size distribution of particles**. *International Journal of Mining, Reclamation and Environment* (37), pages 255–276.

[Fuchs2025] Fuchs, Marco, Blum, Philipp, Bl"ocher, Guido, Scholt‘es, Luc (2025), **Permeability evolution and gouge formation during fracture shearing**. *Geophysical Research Letters* (52), pages e2025GL117217.

[Gao2024a] Gao, Yan, Sun, Ketian, Yuan, Quan, Shi, Tiangen (2024), **Stiffness Anisotropy and Micro-Mechanism of Calcareous Sand with Different Particle Breakage Ratios Subjected to Shearing Based on DEM Simulations**. *Journal of Marine Science and Engineering* (12), pages 702.

[Gao2024b] Gao, Xuesong, Aryan, Aryan, Zhang, Wei (2024), **Numerical Analysis of Rotating Scans' Effect on Surface Roughness in Laser-Powder Bed Fusion**. *Journal of Materials Research and Technology*.

[Gao2025] Gao, Yan, Sun, Ketian, Yuan, Quan, Sun, Le, Tang, Xudong (2025), **Particle Shape-Driven Stiffness Anisotropy in Calcareous Sand and the Underlying Mechanism**. *Applied Sciences* (15), pages 12682.

[Giuliano2023] Giuliano, Lorenzo Vasquez, Buffo, Antonio, Vanni, Marco, Frungieri, Graziano (2023), **Micromechanics and strength of agglomerates produced by spray drying**. *JCIS Open* (9), pages 100068.

[Gladky2017] Gladkyy, Anton, Kuna, Meinhard (2017), **DEM simulation of polyhedral particle cracking using a combined Mohr–Coulomb–Weibull failure criterion**. *Granular Matter* (19), pages 41. DOI 10.1007/s10035-017-0731-8

[Gladky2014] Gladkyy, Anton, Schwarze, Rüdiger (2014), **Comparison of different capillary bridge models for application in the discrete element method**. *Granular Matter*, pages 1–10. DOI 10.1007/s10035-014-0527-z

[Gougucs2020] Göğüş, Özge Dinç (2020), **3D discrete analysis of damage evolution of hard rock under tension**. *Arabian Journal of Geosciences* (13), pages 1–11.

[Grabowski2020] Grabowski, Aleksander, Nitka, Michal (2020), **3D DEM SIMULATIONS OF BASIC GEOTECHNICAL TESTS WITH EARLY DETECTION OF SHEAR LOCALIZATION.**. *Studia Geotechnica et Mechanica*.

[Grabowski2021] Grabowski, A, Nitka, M, Tejchman, J (2021), **Micro-modelling of shear localization during quasi-static confined granular flow in silos using DEM**. *Computers and Geotechnics* (134), pages 104108.

[Grabowski2025] Grabowski, Aleksander (2025), **Analysis of the deformation mechanism of granular material in a direct shear test using the Discrete Element Method**. *Materialy Budowlane*.

[Grujicic2013] Grujicic, M, Snipes, JS, Ramaswami, S, Yavari, R (2013), **Discrete Element Modeling and Analysis of Structural Collapse/Survivability of a Building Subjected to Improvised Explosive Device (IED) Attack**. *Advances in Materials Science and Applications* (2), pages 9–24.

[Guo2014] Guo, Ning, Zhao, Jidong (2014), **A coupled FEM/DEM approach for hierarchical multiscale modelling of granular media**. *International Journal for Numerical Methods in Engineering* (99), pages 789–818. DOI 10.1002/nme.4702

[Guo2015] Guo N., J. Zhao (2015), **Multiscale insights into classical geomechanics problems**. *International Journal for Numerical and Analytical Methods in Geomechanics*. (under review)

[Guo2025] Guo, Chang, Zhou, Chao, Meguid, Mohamed A (2025), **Effect of surface roughness on axial soil-pipe interaction: a discrete element approach**. *Computers and Geotechnics* (187), pages 107464.

[Gusenbauer2018] Gusenbauer Markus, Thomas Schrefl (2018), **Simulation of magnetic particles in microfluidic channels**. *Journal of Magnetism and Magnetic Materials* (446), pages 185–191. DOI 10.1016/j.jmmm.2017.09.031

[Gusenbauer2012] Gusenbauer, M., Kovacs, A., Reichel, F., Exl, L., Bance, S., Özelt, H., Schrefl, T. (2012), **Self-organizing magnetic beads for biomedical applications**. *Journal of Magnetism and Magnetic Materials* (324), pages 977–982.

[Gusenbauer2014] Gusenbauer, M., Nguyen, H., Reichel, F., Exl, L., Bance, S., Fischbacher, J., Özelt, H., Kovacs, A., Brandl, M., Schrefl, T. (2014), **Guided self-assembly of magnetic beads for biomedical applications**. *Physica B: Condensed Matter* (435), pages 21–24.

[Hadda2015] Hadda, N., Nicot, F., Wan, R., Darve, F. (2015), **Microstructural self-organization in granular materials during failure**. *Comptes Rendus Mécanique*.

[Hadda2013] Hadda, Nejib, Nicot, François, Bourrier, Franck, Sibille, Luc, Radjai, Farhang, Darve, Félix (2013), **Micromechanical analysis of second order work in granular media**. *Granular Matter* (15), pages 221–235. DOI 10.1007/s10035-013-0402-3

[Harthong2012b] Harthong, B., Jerier, J.-F., Richefeu, V., Chareyre, B., Doremus, P., Imbault, D., Donzé, F.V. (2012), **Contact impingement in packings of elastic–plastic spheres, application to powder compaction**. *International Journal of Mechanical Sciences* (61), pages 32–43.

[Harthong2009] Harthong, B., Jerier, J.F., Doremus, P., Imbault, D., Donzé, F.V. (2009), **Modeling of high-density compaction of granular materials by the Discrete Ele-**

ment Method. *International Journal of Solids and Structures* (46), pages 3357–3364. DOI 10.1016/j.ijsolstr.2009.05.008

[Hartmann2022] Hartmann, Philipp, Thoeni, Klaus, Rojek, Jerzy (2022), **A generalised multi-scale Peridynamics–DEM framework and its application to rigid–soft particle mixtures**. *Computational Mechanics*, pages 1–20.

[Hartong2012a] Harthong, B., Scholtès, L., Donzé, F.-V. (2012), **Strength characterization of rock masses, using a coupled DEM–DFN model**. *Geophysical Journal International* (191), pages 467–480. DOI 10.1111/j.1365-246X.2012.05642.x

[Hasan2016] Hasan, Alsidqi, Karrech, Ali, Chareyre, Bruno (2016), **Evaluating Force Distributions within Virtual Uncemented Mine Backfill Using Discrete Element Method**. *International Journal of Geomechanics*, pages 06016042.

[Hassan2021] Hassan, Nadine Ali, Nguyen, Ngoc Son, Marot, Didier, Bendahmane, Fateh (2021), **Effect of Scalping on the Mechanical Behavior of Coarse Soils**. *International Journal of Geotechnical and Geological Engineering* (15), pages 64–74.

[Haustein2017] Haustein Martin, Anton Gladkyy, Rüdiger Schwarze (2017), **Discrete element modeling of deformable particles in YADE**. *SoftwareX* (6), pages 118–123. DOI https://doi.org/10.1016/j.softx.2017.05.001 ()

[He2021] He, Hantao, Zheng, Junxing, Li, Zhaochao (2021), **Accelerated simulations of direct shear tests by physics engine**. *Computational Particle Mechanics* (8), pages 471–492.

[Heider2021] Heider, Yousef, Suh, Hyoung Suk, Sun, WaiChing (2021), **An offline multi-scale unsaturated poromechanics model enabled by self-designed/self-improved neural networks**. *International Journal for Numerical and Analytical Methods in Geomechanics*.

[Hilton2013] Hilton, J. E., Tordesillas, A. (2013), **Drag force on a spherical intruder in a granular bed at low Froude number**. *Phys. Rev. E* (88), pages 062203. DOI 10.1103/PhysRevE.88.062203

[Horvath2022] Horv'ath, D'aniel, Tam'as, Kornél, Po'os, Tibor (2022), **Viscoelastic contact model development for the discrete element simulations of mixing process in agitated drum**. *Powder Technology* (397), pages 117038.

[Hosseinkhani2023] Hosseinkhani, Elham, Habibagahi, Ghassem, Nikooee, Ehsan (2023), **Cyclic modeling of unsaturated sands using a pore-scale hydromechanical approach**. *International Journal for Numerical and Analytical Methods in Geomechanics* (47), pages 457–481.

[Houlsby2009] Houlsby G.T. (2009), **Potential particles: a method for modelling non-circular particles in DEM**. *Computers and Geotechnics* (36), pages 953–959. DOI 10.1016/j.compgeo.2009.03.001

[Hu2022] Hu, Z, Yang, ZX, Guo, N, Zhang, YD (2022), **Multiscale modeling of seepage-induced suffusion and slope failure using a coupled FEM–DEM approach**. *Computer Methods in Applied Mechanics and Engineering* (398), pages 115177.

[Huber2024] Marius Huber, Luc Scholtès, Jérôme Lavé (2024), **Stability and failure modes of slopes with anisotropic strength: Insights from discrete element models**. *Geomorphology* (444), pages 108946. DOI https://doi.org/10.1016/j.geomorph.2023.108946

[Hung2023] Hung, Chien-Cheng, Niemeijer, André R, Raoof, Amir, Sweijen, Thomas (2023), **Investigation of strain localization in sheared granular layers using 3-D discrete element modeling**. *Tectonophysics*, pages 229974.

[Ita2023] Ita, Paola, Santa-Cruz, Sandra, Daudon, Dominique, Tarque, Nicola, P'arraga, Anghie, Ramos, Vladimir (2023), **Out-of-plane analysis of dry-stone walls using a pseudo-static experimental and numerical approach in natural-scale specimens**. *Engineering Structures* (288), pages 116153.

[Ivannikov2022] Ivannikov, V, Thomsen, F, Ebel, T, Willumeit–R"omer, R (2022), **Coupling the discrete element method and solid state diffusion equations for modeling of metallic powders sintering**. *Computational Particle Mechanics*, pages 1–23.

[Jasik2018] Jasik P., J. Kozicki, T. Kilich, J.E. Sienkiewicz, N. E. Henriksen (2018), **Electronic structure and rovibrational predissociation of the $2^1\Pi$ state in KLi**. *Physical Chemistry Chemical Physics* (20), pages 18663–18670. DOI 10.1039/c8cp02551g

[Jerier2010b] Jerier, J.-F., Hathong, B., Richefeu, V., Chareyre, B., Imbault, D., Donzé, F.-V., Doremus, P. (2010), **Study of cold powder compaction by using the discrete element method**. *Powder Technology* (In Press). DOI 10.1016/j.powtec.2010.08.056

[Jerier2009] Jerier, J.-F., Imbault, D.and Donzé, F.V., Doremus, P. (2009), **A geometric algorithm based on tetrahedral meshes to generate a dense polydisperse sphere packing**. *Granular Matter* (11). DOI 10.1007/s10035-008-0116-0

[Jerier2010a] Jerier, J.-F., Richefeu, V., Imbault, D., Donzé, F.V. (2010), **Packing spherical discrete elements for large scale simulations**. *Computer Methods in Applied Mechanics and Engineering*. DOI 10.1016/j.cma.2010.01.016

[Jiang2023a] Jiang, Xiao-Qiong, Liu, En-Long (2023), **Evolution of the force chain structure of partially saturated granular material under triaxial compression conditions**. *Computers and Geotechnics* (157), pages 105335.

[Jiao2023b] Jiao, Liqing, Tapponnier, Paul, Donzé, Frédéric-Victor, Scholt'es, Luc, Gaudemer, Yves, Xu, Xiwei (2023), **Discrete element modeling of Southeast Asia's 3D lithospheric deformation during the Indian collision**. *Journal of Geophysical Research: Solid Earth* (128), pages e2022JB025578.

[Jiao2024] Jiao, Liqing, Tapponnier, Paul, Coudurier-Curveur Mccallum, Aurélie, Xu, Xiwei (2024), **The shape of the Himalayan "Arc": An ellipse pinned by syntaxial strike-slip fault tips**. *Proceedings of the National Academy of Sciences* (121), pages e2313278121.

[Jiao2025] Jiao, Liqing, Jiao, Yang, Zhang, Yueqiao (2025), **Controlling factors of the strike-slip fault segmentation: insight from DEM modelling**. *Journal of Structural Geology*, pages 105586.

[Jiaxin2024] Jiaxin, Liu, Tian, Yang, Wang, Zhongkui, Li, Longchuan, Ma, Shugen (2024), **Modeling the impact of terrain surface deformation on drag force using discrete element method and empirical formulation**. *Applied Mathematical Modelling*, pages 115636.

[Jin2024] Jin, Ziyu, Liu, Jiaying, Sun, Honglei, Sun, Miaomiao, Xu, Xiaorong (2024), **Influence of gradation range on strong contact network in granular materials**. *Granular Matter* (26), pages 37.

[Jin2025] Jin, Ziyu, Liu, Jiaying, Ma, Gang, Hu, Chengbao, Yang, Qihang, Shi, Xiusong, Wang, Xinquan (2025), **How does the largest cluster in the strong network rule granular soil mechanics? A DEM study**. *International Journal for Numerical and Analytical Methods in Geomechanics* (49), pages 839–859.

[Kalogeropoulos2022] Kalogeropoulos A.D , Michalakopoulos T.N (2022), **The effect of grain interlocking in discrete element modelling of rock cutting**. *Geomechanics and Geoengineering*, pages 1–24. DOI 10.1080/17486025.2022.2064553

[Kalogeropoulos2022b] A.D Kalogeropoulos, T.N Michalakopoulos (2022), **A Method for Selecting Optimum Microparameters' Values in the Numerical Simulation of Rock Cutting**. *Mining, Metallurgy & Exploration*. DOI 10.1007/s42461-022-00724-8

[Kaufmann2023] Kaufmann, Georg, Romanov, Douchko, Werban, Ulrike, Vienken, Thomas (2023), **The M"unsterdorf sinkhole cluster: void origin and mechanical failure**. *Solid Earth* (14), pages 333–351.

[Kozicki2012] Kozicki J., J. Tejchman, Z. Mróz (2012), **Effect of grain roughness on strength, volume changes, elastic and dissipated energies during quasi-static homogeneous triaxial compression using DEM.**. *Granular Matter* (14), pages 457–468. DOI 10.1007/s10035-012-0352-1

[Kozicki2008] Kozicki, J., Donzé, F.V. (2008), **A new open-source software developed for numerical simulations using discrete modeling methods**. *Computer Methods in Applied Mechanics and Engineering* (197), pages 4429–4443. DOI 10.1016/j.cma.2008.05.023

[Kozicki2009] Kozicki, J., Donzé, F.V. (2009), **YADE-OPEN DEM: an open-source software using a discrete element method to simulate granular material**. *Engineering Computations* (26), pages 786–805. DOI 10.1108/02644400910985170

[Kozicki2006a] Kozicki, J., Tejchman, J. (2006), **2D Lattice Model for Fracture in Brittle Materials**. *Archives of Hydro-Engineering and Environmental Mechanics* (53), pages 71–88.

[Kozicki2007a] Kozicki, J., Tejchman, J. (2007), **Effect of aggregate structure on fracture process in concrete using 2D lattice model"**. *Archives of Mechanics* (59), pages 365–384.

[Kozicki2016] Kozicki, J., Tejchman, J. (2016), **DEM investigations of two-dimensional granular vortex- and anti-vortex structures during plane strain compression.**. *Granular Matter* (18). DOI 10.1007/s10035-016-0627-z

[Kozicki2017] Kozicki, J., Tejchman, J. (2017), **Investigations of quasi-static vortex structures in 2D sand specimen under passive earth pressure conditions based on DEM and Helmholtz-Hodge vector field decomposition**. *Granular Matter* (19), pages 31. DOI 10.1007/s10035-017-0714-9

[Kozicki2018] Kozicki, J., Tejchman, J. (2018), **Relationship between vortex structures and shear localization in 3D granular specimens based on combined DEM and Helmholtz-Hodge decomposition**. *Granular Matter* (20), pages 48. DOI 10.1007/s10035-018-0815-0

[Kozicki2014] Kozicki, J., Tejchman, Jacek, Mühlhaus, Hans-Bernd (2014), **Discrete simulations of a triaxial compression test for sand by DEM**. *International Journal for Numerical and Analytical Methods in Geomechanics* (38), pages 1923–1952. DOI 10.1002/nag.2285

[Kozicki2022] Kozicki, Janek, Gladky, Anton, Thoeni, Klaus (2022), **Implementation of high-precision computation capabilities into the open-source dynamic simulation framework YADE**. *Computer Physics Communications* (270), pages 108167. DOI 10.1016/j.cpc.2021.108167

[Kozicki2019] Krzaczek, M., Nitka, M., Kozicki, J., Tejchman, J. (2019), **Simulations of hydro-fracking in rock mass at meso-scale using fully coupled DEM/CFD approach**. *Acta Geotechnica*. DOI 10.1007/s11440-019-00799-6

[Krzaczek2021] Krzaczek, Marek, Nitka, Michal, Tejchman, J (2021), **Effect of gas content in macro-pores on hydraulic fracturing in rocks using a fully coupled DEM/CFD approach**. *International Journal for Numerical and Analytical Methods in Geomechanics* (45), pages 234–264.

[Krzaczek2023] Krzaczek, M, Tejchman, J (2023), **Hydraulic fracturing process in rocks–small-scale simulations with a novel fully coupled DEM/CFD-based thermo-hydro-mechanical approach**. *Engineering Fracture Mechanics*, pages 109424.

[Krzaczek2024a] Krzaczek, Marek, Tejchman, J, Nitka, Michal (2024), **Effect of free water on the quasi-static compression behavior of partially-saturated concrete with a fully coupled DEM/CFD approach**. *Granular Matter* (26), pages 38.

[Krzaczek2024b] Krzaczek, Marek, Tejchman, J, Nitka, Michal (2024), **Coupled DEM/CDF analysis of impact of free water on the static and dynamic response of concrete in tension regime**. *Computers and Geotechnics* (172), pages 106449.

[Krzaczek2025] Krzaczek, Marek, Nitka, Michal, Tejchman, Jacek (2025), **Impact of strain rate, free water, and aggregate fragmentation on the dynamic behavior of concrete in compression regime using a unique coupled DEM/CFD technique**. *Granular Matter* (27), pages 79.

[Kunhappan2017] Kunhappan, D, Harthong, B, Chareyre, B, Balarac, G, Dumont, PJJ (2017), **Numerical modeling of high aspect ratio flexible fibers in inertial flows**. *Physics of Fluids*. DOI 10.3390/min7040056

[Lambert2025] Lambert, Clovis, Maurin, Rapha"el, Lacaze, Laurent, Fede, Pascal (2025), **Combined influence of particle friction and inertia on hysteresis in granular media on an inclined plane**. *Physical Review Fluids* (10), pages 034301.

[Lapcevic2017] Lapčević, Veljko, Torbica, Slavko (2017), **Numerical Investigation of Caved Rock Mass Friction and Fragmentation Change Influence on Gravity Flow Formation in Sublevel Caving**. *Minerals* (7). DOI 10.3390/min7040056

[Larijani2025] Larijani, Roxana Saghafian, Magnanimo, Vanessa, Luding, Stefan (2025), **A Coarse-Grained Discrete Element Model (CG-DEM) based on parameter scaling for dense wet granular system**. *Powder Technology* (453), pages 120581.

[Li2021] Li, Guang-yao, Zhan, Liang-tong, Hu, Zheng, Chen, Yun-min (2021), **Effects of particle gradation and geometry on the pore characteristics and water retention curves of granular soils: a combined DEM and PNM investigation**. *Granular Matter* (23), pages 1–16.

[Li2023] Li, Weichao, Chu, Yifan, Deng, Gang, Cai, Hong, Xie, Dingsong, Lee, Min Lee (2023), **Study of shear induced stress redistribution in gap-graded soils by discrete element method**. *Computers and Geotechnics* (156), pages 105248.

[Li2024] Li, Xin, Kouretzis, George, Thoeni, Klaus (2024), **Discrete Element Modelling of uplift of rigid pipes deeply buried in dense sand**. *Computers and Geotechnics* (166), pages 105957.

[Li2025a] Li, Hanze, Zhao, Chaofa, Argilaga, Albert, Chen, Yanni (2025), **Effects of water content on the evolution of capillary stress in unsaturated granular soils: DEM simulations**. *Powder Technology*, pages 121095.

[Li2025b] Li, Xin, Kouretzis, George, Thoeni, Klaus (2025), **Scale effects on lateral soil-buried pipe interaction**. *Acta Geotechnica*, pages 1–17.

[Li2025] Li, Xin, Kouretzis, George, Thoeni, Klaus (2025), **Enhanced calibration of Discrete Element models for soil-structure interaction problems**. *Computers and Geotechnics* (188), pages 107539.

[Liang2023] Liang, Chenguang, Yin, Yan, Wang, Wenxuan, Yi, Min (2023), **A thermodynamically consistent non-isothermal phase-field model for selective laser sintering**. *International Journal of Mechanical Sciences*, pages 108602.

[Ling2025] Ling, Bowen, Du, Xingming, Du, Shuheng, Yang, Fengchang (2025), **A novel numerical simulation framework for predicting pore space evolution and rock properties through sedimentation, compaction and diagenesis**. *Scientific Reports* (15), pages 17171.

[Liu2022] Liu, Jiaying, Wautier, Antoine, Nicot, Franccois, Darve, Félix, Zhou, Wei (2022), **How meso shear chains bridge multiscale shear behaviors in granular materials: a preliminary study**. *International Journal of Solids and Structures* (252), pages 111835.

[Liu2024] Liu, Xin, Zhou, Annan, Wang, Xiaonan, Shen, Shui-Long (2024), **A fully coupled micro-hydromechanical (micro-HM) model for partially saturated soils based on DEM**. *Computers and Geotechnics* (173), pages 106531.

[Liu2025a] Liu, Feng, Tang, Hongxiang, Shahin, Mohamed A, Zhao, Honghua, Karrech, Ali, Zhu, Feng, Zhou, He (2025), **Multiscale simulation study for mechanical characteristics of coral sand influenced by particle breakage**. *Powder Technology* (449), pages 120387.

[Liu2025b] Liu, Chuanqi, Ru, Min, Li, Zonghan, Liang, Jinding (2025), **A numerical model for the electrode calendering process considering the cohesive behavior between damageable particles and metal foil**. *Computational Particle Mechanics* (12), pages 3221–3234.

[Lombardo2025] Lombardo Pontillo, Alessio, Marcato, Agnese, Versaci, Daniele, Marchisio, Daniele, Boccardo, Gianluca (2025), **Comparative Analysis via CFD Simulation on the Impact of Graphite Anode Morphologies on the Discharge of a Lithium-Ion Battery**. *Batteries* (11), pages 252.

[Lomine2013] Lominé, F., Scholtès, L., Sibille, L., Poullain, P. (2013), **Modelling of fluid-solid interaction in granular media with coupled LB/DE methods: application to piping erosion**. *International Journal for Numerical and Analytical Methods in Geomechanics* (37), pages 577–596. DOI 10.1002/nag.1109

[Mahmoodlu2016] Mahmoodlu, MG, Raoof, A, Sweijen, T, van Genuchten, M TH (2016), **Effects of Sand Compaction and Mixing on Pore Structure and the Unsaturated Soil Hydraulic Properties**. *Vadose Zone* (15). DOI 10.2136/vzj2015.10.0136

[Marcato2022] Marcato, Agnese, Boccardo, Gianluca, Marchisio, Daniele (2022), **From Computational Fluid Dynamics to Structure Interpretation via Neural Networks: An Application to Flow and Transport in Porous Media**. *Industrial & Engineering Chemistry Research*.

[Mariani2021] Mariani, Marco, Beltrami, Ruben, Brusa, Paolo, Galassi, Carmen, Ardito, Raffaele, Lecis, Nora (2021), **3D printing of fine alumina powders by binder jetting**. *Journal of the European Ceramic Society*.

[Marzougui2015] Marzougui, Donia, Chareyre, Bruno, Chauchat, Julien (2015), **Microscopic origins of shear stress in dense fluid–grain mixtures**. *Granular Matter*, pages 1–13. DOI 10.1007/s10035-015-0560-6

[Massoumi2023] Massoumi, Sina, Challamel, No"el, Lerbet, Jean, Wautier, Antoine, Nicot, Franccois, Darve, Félix (2023), **Shear vibration modes of granular structures: Continuous and discrete approaches**. *ZAMM-Journal of Applied Mathematics and Mechanics/Zeitschrift f"ur Angewandte Mathematik und Mechanik* (103), pages e202200391.

[Maurin2015b] Maurin, R., Chauchat, J., Chareyre, B., Frey, P. (2015), **A minimal coupled fluid-discrete element model for bedload transport**. *Physics of Fluids* (27), pages 113302. DOI 10.1063/1.4935703

[Maurin2016] Maurin, R., Chauchat, J., Frey, P. (2016), **Dense granular flow rheology in turbulent bedload transport**. *Journal of Fluid Mechanics* (804), pages 490–512. DOI 10.1017/jfm.2016.520

[Maurin2018] Maurin, R., Chauchat, J., Frey, P (2018), **Revisiting slope influence in turbulent bedload transport: consequences for vertical flow structure and transport rate scaling**. *Journal of Fluid Mechanics* (839), pages 135–156. DOI 10.1017/jfm.2017.903

[Mede2018] Mede, T., Chambon, G., Hagenmuller, P., Nicot, F. (2018), **A medial axis based method for irregular grain shape representation in DEM simulations**. *Granular Matter* (20), pages 16. DOI 10.1007/s10035-017-0785-7

[Mede2024] Mede, Tijan, Godec, Matjavz (2024), **Relevance of inter-particle interaction in directed energy deposition powder stream**. *Powder Technology* (435), pages 119393.

[MesaAlcantara2024] Mesa-Alcantara, Arisleidy, Romero, Enrique, Torres-Serra, Joel, Mokni, Nadia (2024), **Compressibility of a binary bentonite-based mixture with particular emphasis on pellet orientation**. *Applied Clay Science* (261), pages 107575.

[Montella2016] Montellà, E. P., Toraldo, M., Chareyre, B., Sibille, L. (2016), **Localized fluidization in granular materials: Theoretical and numerical study**. *Phys. Rev. E* 5 (94), pages 052905. DOI 10.1103/PhysRevE.94.052905

[Montella2020] Montellà, Eduard Puig, Yuan, Chao, Chareyre, Bruno, Gens, Antonio (2020), **Hybrid multi-scale model for partially saturated media based on a pore network approach and lattice boltzmann method**. *Advances in Water Resources* (144), pages 103709.

[Montella2020b] Montellà, EP, Chareyre, B, Salager, S, Gens, A (2020), **Benchmark cases for a multicomponent Lattice–Boltzmann method in hydrostatic conditions**. *MethodsX* (7), pages 101090.

[Morimoto2024] Morimoto, Tokio, O'Sullivan, Catherine, Taborda, David MG (2024), **Applying Network Modeling to Determine Seepage-Induced Forces on Soil Particles**. *Journal of Geotechnical and Geoenvironmental Engineering* (150), pages 04024029.

[Mostafa2023] Mostafa, Ahmad, Scholt'es, Luc, Golfier, Fabrice (2023), **Pore-scale hydro-mechanical modeling of gas transport in coal matrix**. *Fuel* (345), pages 128165.

[Munch2024] Munch, Peter, Ivannikov, Vladimir, Cyron, Christian, Kronbichler, Martin (2024), **On the construction of an efficient finite-element solver for phase-field simulations of many-particle solid-state-sintering processes**. *Computational Materials Science* (231), pages 112589.

[Nguyen2021b] Nguyen, Hien Nho Gia, Scholtès, Luc, Guglielmi, Yves, Donzé, Frédéric Victor, Ouraga, Zady, Souley, Mountaka (2021), **Micromechanics of Sheared Granular Layers Activated by Fluid Pressurization**. *Geophysical Research Letters* (48), pages e2021GL093222. DOI 10.1029/2021GL093222 (e2021GL093222 2021GL093222)

[Nguyen2021a] Nguyen, Ngoc-Son, Taha, Habib, Marot, Didier (2021), **A new Delaunay triangulation-based approach to characterize the pore network in granular materials**. *Acta Geotechnica*, pages 1–19.

[Nicot2011] Nicot, F., Hadda, N., Bourrier, F., Sibille, L., Darve, F. (2011), **Failure mechanisms in granular media: a discrete element analysis**. *Granular Matter* (13), pages 255–260. DOI 10.1007/s10035-010-0242-3

[Nicot2013a] Nicot, F., Hadda, N., Darve, F. (2013), **Second-order work analysis for granular materials using a multiscale approach**. *International Journal for Numerical and Analytical Methods in Geomechanics*. DOI 10.1002/nag.2175

[Nicot2013b] Nicot, F., Hadda, N., Guessasma, M., Fortin, J., Millet, O. (2013), **On the definition of the stress tensor in granular media**. *International Journal of Solids and Structures*. DOI 10.1016/j.ijsolstr.2013.04.001

[Nicot2012] Nicot, F., Sibille, L., Darve, F. (2012), **Failure in rate-independent granular materials as a bifurcation toward a dynamic regime**. *International Journal of Plasticity* (29), pages 136–154. DOI 10.1016/j.ijplas.2011.08.002

[Niedostatkiewicz2013] Niedostatkiewicz M., J. Kozicki, J. Tejchman, H.B. Mühlhaus (2013), **Discrete modelling results of a direct shear test for granular materials versus FE results.**. *Granular Matter* (15). DOI 10.1007/s10035-013-0423-y

[Nitka2015b] Nitka M., J. Tejchman, J. Kozicki, D. Leśniewska (2015), **DEM analysis of microstructural events within granular shear zones under passive earth pressure conditions.**. *Granular Matter* (17). DOI 10.1007/s10035-015-0558-0

[Nitka2015a] Nitka, M., Tejchman, J. (2015), **Modelling of concrete behaviour in uniaxial compression and tension with DEM**. *Granular Matter*, pages 1–20.

[Nitka2024] Nitka, Michal, Tejchman, Jacek (2024), **Mesoscopic simulations of a fracture process in reinforced concrete beam in bending using a 2D coupled DEM/micro-CT approach**. *Engineering Fracture Mechanics* (304), pages 110153.

[Nitka2025] Nitka, Michal, Tejchman, Jacek (2025), **Effects of aggregate crushing and strain rate on fracture in compressive concrete with a DEM-based breakage model**. *Granular Matter* (27), pages 8.

[Noel2022] No"el, Emeline, Teixeira, David (2022), **New framework for upscaling gas-solid heat transfer in dense packing**. *International Journal of Heat and Mass Transfer* (189), pages 122745.

[Noel2023] No"el, Emeline, Teixeira, David, Preux, Gauthier (2023), **Modelling of gas-solid heat transfer and pressure drop in a rock-packed bed using pore-scale simulations**. *International Journal of Heat and Mass Transfer* (214), pages 124432.

[Orosz2023] Orosz, 'Akos, Bagi, Katalin (2023), **Comparison of contact treatment methods for rigid polyhedral discrete element models**. *International Journal of Rock Mechanics and Mining Sciences* (170), pages 105550.

[Ostanin2020] Ostanin, Igor A, Oganov, Artem R, Magnanimo, Vanessa (2020), **Collapse modes in simple cubic and body-centered cubic arrangements of elastic beads**. *Physical Review E* (102), pages 032901.

[Pan2022] Pan, Jin-Hong, Zhang, Jian-Min, Wang, Rui (2022), **Influence of small particle surface asperities on macro and micro mechanical behavior of granular material**. *International Journal for Numerical and Analytical Methods in Geomechanics* (46), pages 961–978.

[Papachristos2017] Papachristos, E, Scholtès, L, Donzé, F.V, Chareyre, B (2017), **Intensity and volumetric characterizations of hydraulically driven fractures by hydro-mechanical simulations**. *International Journal of Rock Mechanics and Mining Sciences* (93), pages 163–178.

[Papachristos2023] Papachristos, E, Stefanou, I, Sulem, J (2023), **A discrete elements study of the frictional behavior of fault gouges**. *Journal of Geophysical Research: Solid Earth* (128), pages e2022JB025209.

[Pekmezi2020] Pekmezi, Gerald, Littlefield, David, Chareyre, Bruno (2020), **Statistical distributions of the elastic moduli of particle aggregates at the mesoscale**. *International Journal of Impact Engineering* (139), pages 103481.

[Pekmezi2024] Pekmezi, Gerald, Chareyre, Bruno, Littlefield, David (2024), **Uniform boundary conditions on models of spherical particles through alpha shape surface tracking and Laguerre–Voronoi diagrams**. *Computer Physics Communications* (301), pages 109214.

[Pelech2022] Pelech, T, Barnett, N, Dello-Iacovo, M, Oh, J, Saydam, S (2022), **Analysis of the stability of micro-tunnels in lunar regolith with the Discrete Element Method**. *Acta Astronautica* (196), pages 1–12.

[Pirnia2018] Pirnia Pouyan, François Duhaime, Yannic Ethier, Jean-Sébastien Dubé (2018), **ICY: An interface between COMSOL multiphysics and discrete element code YADE for the modelling of porous media**. *Computers & Geosciences*. DOI https://doi.org/10.1016/j.cageo.2018.11.002

[Pirrone2023] Pirrone, Serena RM, Del Dottore, Emanuela, Sibille, Luc, Mazzolai, Barbara (2023), **A methodology to investigate the design requirements of plant root-inspired robots for soil exploration**. *IEEE Robotics and Automation Letters*.

[Pirrone2025] Pirrone, Serena Rosa Maria, Del Dottore, Emanuela, Just, Gunter, Mazzolai, Barbara, Sibille, Luc (2025), **The effect of tip design on technological performance during the exploration of Earth, Lunar, and Martian soil environments**. *Journal of Field Robotics*.

[Pol2021] Pol, Antonio, Gabrieli, Fabio, Brezzi, Lorenzo (2021), **Discrete element analysis of the punching behaviour of a secured drapery system: from laboratory characterization to idealized in situ conditions**. *Acta Geotechnica*, pages 1–21.

[Pol2022] Pol, Antonio, Gabrieli, Fabio (2022), **Anchor plate bearing capacity in flexible mesh facings**. *Soils and Foundations* (62), pages 101222.

[Pol2025] Pol, Antonio, Storti, Sara, Gabrieli, Fabio (2025), **Granular drag and lift force on a flexible fiber**. *Physical Review E* (112), pages 045425.

[Pouragha2021] Pouragha, Mehdi, Kruyt, Niels P, Wan, Richard (2021), **Non-coaxial Plastic Flow of Granular Materials through Stress Probing Analysis**. *International Journal of Solids and Structures*.

[Puckett2011] Puckett, J.G., Lechenault, F., Daniels, K.E. (2011), **Local origins of volume fraction fluctuations in dense granular materials**. *Physical Review E* (83), pages 041301. DOI 10.1103/PhysRevE.83.041301

[Redaelli2021] Redaelli, Irene, di Prisco, Claudio (2021), **DEM numerical tests on dry granular specimens: the role of strain rate under evolving/unsteady conditions**. *Granular Matter* (23), pages 1–34.

[Reiner2023] Reiner, Johannes, Nguyen, Nhu HT (2023), **Meshfree simulation of progressive damage in composite laminates using discrete element analysis**. *Journal of Composite Materials* (57), pages 1135–1148.

[Reshetnyk2025] Reshetnyk, Volodymyr, Luk'yanyk, Igor, Skorov, Yuri, Grynko, Yevgen, Macher, Wolfgang, Schuckart, Christian, Zhao, Yuhui, Blum, J"urgen (2025), **Key structural characteristics of porous layers in diffusion modelling: A study on polydispersity, shape, and hierarchy**. *Planetary and Space Science*, pages 106078.

[Rioual2024] Rioual, Franccois, Gbehe, Paule Emmanuelle Eva (2024), **Characterisation of the granular dynamics at the interface between a pipe and a granular flow in a rotating drum**. *Particuology* (86), pages 117–125.

[Rojas2023] Rojas, Eduardo, Alarc'on, Héctor, Salinas, Vicente, Castillo, Gustavo, Gutiérrez, Pablo (2023), **Stability of a tilted granular monolayer: How many spheres can we pick before the collapse?**. *Physical Review E* (108), pages 064904.

[SacMorane2025] Sac-Morane, Alexandre, Rattez, Hadrien, Veveakis, Manolis (2025), **Importance of Precipitation in the Slowdown of Creep Behavior Induced by Pressure Solution**. *Journal of Engineering Mechanics* (151), pages 04025025.

[Salomon2024] Salomon, Jose, O'Sullivan, Catherine, Patino-Ramirez, Fernando (2024), **On data benchmarking and verification of discrete granular simulations**. *Data in Brief* (53), pages 110252.

[Saomoto2023] Saomoto, Hidetaka, Kikkawa, Naotaka, Moriguchi, Shuji, Nakata, Yukio, Otsubo, Masahide, Angelidakis, Vasileios, Cheng, Yi Pik, Chew, Kevin, Chiaro, Gabriele, Duriez, Jérôme, others (2023), **Round robin test on angle of repose: DEM simulation results collected from 16 groups around the world**. *Soils and Foundations* (63), pages 101272.

[Sarkis2022] Sarkis, Marilyn, Abbas, Mohammad, Naillon, Antoine, Emeriault, Fabrice, Geindreau, Christian, Esnault-Filet, Annette (2022), **DEM modeling of biocemented sand: Influence of the cohesive contact surface area distribution and the percentage of cohesive contacts**. *Computers and Geotechnics* (149), pages 104860.

[Sayeed2011] Sayeed, M.A., Suzuki, K., Rahman, M.M., Mohamad, W.H.W., Razlan, M.A., Ahmad, Z., Thumrongvut, J., Seangatith, S., Sobhan, MA, Mofiz, SA, others (2011), **Strength and Deformation Characteristics of Granular Materials under Extremely Low to High Confining Pressures in Triaxial Compression**. *International Journal of Civil & Environmental Engineering IJCEE-IJENS* (11).

[Scholtes2015a] Scholtès, L., Chareyre, B., Michallet, H., Catalano, E., Marzougui, D. (2015), **Modeling wave-induced pore pressure and effective stress in a granular seabed**. *Continuum Mechanics and Thermodynamics* (27), pages 305–323. DOI http://dx.doi.org/10.1007/s00161-014-0377-2

[Scholtes2009a] Scholtès, L., Chareyre, B., Nicot, F., Darve, F. (2009), **Micromechanics of granular materials with capillary effects**. *International Journal of Engineering Science* (47), pages 64–75. DOI 10.1016/j.ijengsci.2008.07.002

[Scholtes2009c] Scholtès, L., Chareyre, B., Nicot, F., Darve, F. (2009), **Discrete modelling of capillary mechanisms in multi-phase granular media**. *Computer Modeling in Engineering and Sciences* (52), pages 297–318.

[Scholtes2015b] Scholtès, L., Donzé, F., V. (2015), **A DEM analysis of step-path failure in jointed rock slopes**. *Comptes rendus - Mécanique* (343), pages 155–165. DOI http://dx.doi.org/10.1016/j.crme.2014.11.002

[Scholtes2011] Scholtès, L., Donzé, F.V., Khanal, M. (2011), **Scale effects on strength of geomaterials, case study: Coal**. *Journal of the Mechanics and Physics of Solids* (59), pages 1131–1146. DOI 10.1016/j.jmps.2011.01.009

[Scholtes2012] Scholtès, L., Donzé, F.V. (2012), **Modelling progressive failure in fractured rock masses using a 3D Discrete Element Method**. *International Journal of Rock Mechanics and Mining Sciences* (52), pages 18–30. DOI 10.1016/j.ijrmms.2012.02.009

[Scholtes2013] Scholtès, L., Donzé, F.V. (2013), **A DEM model for soft and hard rocks: Role of grain interlocking on strength**. *Journal of the Mechanics and Physics of Solids* (61), pages 352–369. DOI 10.1016/j.jmps.2012.10.005

[Scholtes2009b] Scholtès, L., Hicher, P.-Y., Chareyre, B., Nicot, F., Darve, F. (2009), **On the capillary stress tensor in wet granular materials**. *International Journal for Numerical and Analytical Methods in Geomechanics* (33), pages 1289–1313. DOI 10.1002/nag.767

[Scholtes2010] Scholtès, L., Hicher, P.-Y., Sibille, L. (2010), **Multiscale approaches to describe mechanical responses induced by particle removal in granular materials**. *Comptes Rendus Mécanique* (338), pages 627–638. DOI 10.1016/j.crme.2010.10.003

[Senanayake2022] Senanayake, SMCU, Haque, A, Bui, HH (2022), **An experiment-based cohesive-frictional constitutive model for cemented materials**. *Computers and Geotechnics* (149), pages 104862.

[Shafabakhsh2024] Shafabakhsh, Paiman, Le Borgne, Tanguy, Renard, Franccois, Linga, Gaute (2024), **Resolving pore-scale concentration gradients for transverse mixing and reaction in porous media**. *Advances in Water Resources*, pages 104791.

[Shiu2008] Shiu, W., Donzé, F.V., Daudeville, L. (2008), **Compaction process in concrete during missile impact: a DEM analysis**. *Computers and Concrete* (5), pages 329–342.

[Shiu2009] Shiu, W., Donzé, F.V., Daudeville, L. (2009), **Discrete element modelling of missile impacts on a reinforced concrete target**. *International Journal of Computer Applications in Technology* (34), pages 33–41.

[Sibille2015] Sibille, L., Hadda, N., Nicot, F., Tordesillas, A., Darve, F. (2015), **Granular plasticity, a contribution from discrete mechanics**. *Journal of the Mechanics and Physics of Solids* (75), pages 119–139. DOI 10.1016/j.jmps.2014.09.010

[Sibille2014] Sibille, L., Lominé, F., Poullain, P., Sail, Y., Marot, D. (2014), **Internal erosion in granular media: direct numerical simulations and energy interpretation**. *Hydrological Processes*. DOI 10.1002/hyp.10351 (First published online Oct. 2014)

[Singh2024a] Singh, Shubjot, Buscarnera, Giuseppe (2024), **Examining the adaptive elastic anisotropy of granular materials**. *Géotechnique*, pages 1–40.

[Singh2024b] Singh, Shubjot, Buscarnera, Giuseppe (2024), **Deciphering how the particle shape modulates the elastic anisotropy of granular media**. *Computers and Geotechnics* (176), pages 106773.

[Skorov2022] Skorov, Yu, Reshetnyk, V, Bentley, MS, Rezac, Ladislav, Hartogh, Paul, Blum, J (2022), **The effect of hierarchical structure of the surface dust layer on the modelling of comet gas production**. *Monthly Notices of the Royal Astronomical Society* (510), pages 5520–5534.

[Smilauer2006] Šmilauer Václav (2006), **The splendors and miseries of Yade design**. *Annual Report of Discrete Element Group for Hazard Mitigation.*

[Sobieski2020] Sobieski, Wojciech (2020), **Calculating the Binary Tortuosity in DEM-Generated Granular Beds**. *Processes* (8), pages 1105.

[Soundaranathan2023] Soundaranathan, Mithushan, Al-Sharabi, Mohammed, Sweijen, Thomas, Bawuah, Prince, Zeitler, J Axel, Hassanizadeh, S Majid, Pitt, Kendal, Johnston, Blair F, Markl, Daniel (2023), **Modelling the evolution of pore structure during the disintegration of pharmaceutical tablets**. *Pharmaceutics* (15), pages 489.

[Suchorzewski2022] Suchorzewski, Jan, Nitka, Michal (2022), **Size effect at aggregate level in microCT scans and DEM simulation–Splitting tensile test of concrete**. *Engineering Fracture Mechanics* (264), pages 108357.

[Suh2024] Suh, Hyoung Suk (2024), **Evolution of anisotropic capillarity in unsaturated granular media within the pendular regime**. *International Journal of Geo-Engineering* (15), pages 10.

[Suhr2016a] Suhr Bettina, Six Klaus (2016), **On the effect of stress dependent interparticle friction in direct shear tests**. *Powder Technology* (294), pages 211–220. DOI 10.1016/j.powtec.2016.02.029

[Suhr2020] Suhr Bettina, Six Klaus (2020), **Simple particle shapes for DEM simulations of railway ballast – influence of shape descriptors on packing behaviour**. *Granular Matter* (22). DOI 10.1007/s10035-020-1009-0

[Suhr2016b] Suhr, Bettina, Six, Klaus (2016), **Friction phenomena and their impact on the shear behaviour of granular material**. *Computational Particle Mechanics*. DOI 10.1007/s40571-016-0119-2

[Suhr2017] Suhr, Bettina, Six, Klaus (2017), **Parametrisation of a DEM model for railway ballast under different load cases**. *Granular Matter* (19), pages 64. DOI 10.1007/s10035-017-0740-7

[Suhr2022] Suhr, Bettina, Skipper, William A, Lewis, Roger, Six, Klaus (2022), **DEM modelling of railway ballast using the Conical Damage Model: a comprehensive parametrisation strategy**. *Granular Matter* (24), pages 1–25.

[Suhr2022a] Bettina Suhr, Klaus Six (2022), **Efficient DEM simulations of railway ballast using simple particle shapes**. *Granular Matter* (24). DOI https://doi.org/10.1007/s10035-022-01274-y

[Suhr2024] Bettina Suhr, William A. Skipper, Roger Lewis, Klaus Six (2024), **DEM simulation of single sand grain crushing in sanded wheel-rail contacts**. *Powder Technology* (432), pages 119150. DOI https://doi.org/10.1016/j.powtec.2023.119150

[Suhr2025] Suhr, Bettina, Skipper, William A, Lewis, Roger, Six, Klaus (2025), **Mechanisms of adhesion increase in wet sanded wheel–rail contacts—a DEM-based analysis**. *Lubricants* (13), pages 314.

[Sweijen2017b] Sweijen, T, Aslannejad, H, Hassanizadeh, SM (2017), **Capillary pressure-saturation relationships for porous granular materials: Pore morphology method vs. pore unit assembly method**. *Advances in Water Resources* (107), pages 22–31. DOI 10.1016/j.advwatres.2017.06.001

[Sweijen2017] Sweijen, T, Chareyre, B, Hassanizadeh, SM, Karadimitriou, NK (2017), **Grain-scale modelling of swelling granular materials; application to super absorbent polymers**. *Powder Technology* (318), pages 411–422. DOI 10.1016/j.powtec.2017.06.015

[Sweijen2016] Sweijen, T, Nikooee, E, Hassanizadeh, S.M, Chareyre, B (2016), **The effects of swelling and porosity change on capillarity: DEM coupled with a pore-unit assembly method**. *Transport in porous media* (113), pages 207–226. DOI 10.1007/s11242-016-0689-8

[Sweijen2018] Sweijen, Thomas, Hassanizadeh, S Majid, Chareyre, Bruno, Zhuang, Luwen (2018), **Dynamic Pore-Scale Model of Drainage in Granular Porous Media: The Pore-Unit Assembly Method**. *Water Resources Research* (54), pages 4193–4213. DOI 10.1029/2017WR021769

[Sweijen2020] Sweijen, Thomas, Hassanizadeh, S Majid, Chareyre, Bruno (2020), **Unsaturated flow in a packing of swelling particles; a grain-scale model**. *Advances in Water Resources*, pages 103642.

[Tamas2024] Tam'as, Kornél (2024), **Modelling the interaction of soil with a passively-vibrating sweep using the discrete element method**. *Biosystems Engineering* (245), pages 199–222.

[Tedesco2023] Tedesco, Bruna Mota Mendes Silva, Cord~ao Neto, Manoel Porfirio, Farias, M'arcio Muniz de, Tarantino, Alessandro (2023), **Design of agglomerates using Weibull distribution to simulate crushable particles in the discrete element method**. *Soils and Rocks* (46), pages e2023004922.

[Tejada2016] Tejada I. G., L. Sibille, B. Chareyre (2016), **Role of blockages in particle transport through homogeneous granular assemblies**. *EPL (Europhysics Letters)* (115), pages 54005.

[Thoeni2014] Thoeni K., A. Giacomini, C. Lambert, S.W. Sloan, J.P. Carter (2014), **A 3D discrete element modelling approach for rockfall analysis with drapery systems**. *International Journal of Rock Mechanics and Mining Sciences* (68), pages 107–119. DOI 10.1016/j.ijrmms.2014.02.008

[Thoeni2013] Thoeni K., C. Lambert, A. Giacomini, S.W. Sloan (2013), **Discrete modelling of hexagonal wire meshes with a stochastically distorted contact model**. *Computers and Geotechnics* (49), pages 158–169. DOI 10.1016/j.compgeo.2012.10.014

[Tian2020] Tian, Yunfu, Yang, Lijun, Zhao, Dejin, Huang, Yiming, Pan, Jiajing (2020), **Numerical analysis of powder bed generation and single track forming for selective laser melting of SS316L stainless steel**. *Journal of Manufacturing Processes* (58), pages 964–974.

[Tian2023] Tian, Zhiguo, Zhang, Duzhou, Zhou, Gang, Zhang, Shaohua, Wang, Moran (2023), **Compaction and sintering effects on scaling law of permeability-porosity relation of powder materials**. *International Journal of Mechanical Sciences*, pages 108511.

[Tomporowski2022] Tomporowski, D, Nitka, M, Tejchman, J (2022), **Application of the 3D DEM in the modelling of fractures in pre-flawed marble specimens during uniaxial compression**. *Engineering Fracture Mechanics*, pages 108978.

[Tong2012] Tong, A.-T., Catalano, E., Chareyre, B. (2012), **Pore-Scale Flow Simulations: Model Predictions Compared with Experiments on Bi-Dispersed Granular Assemblies**. *Oil & Gas Science and Technology - Rev. IFP Energies nouvelles*. DOI 10.2516/ogst/2012032

[Tordesillas2020] Tordesillas, Antoinette, Kahagalage, Sanath, Ras, Charl, Nitka, Michal, Tejchman, Jacek (2020), **Early prediction of macrocrack location in concrete, rocks and other granular composite materials**. *Scientific reports* (10), pages 1–16.

[Tran2012] Tran, V.D.H., Meguid, M.A., Chouinard, L.E. (2012), **An Algorithm for the Propagation of Uncertainty in Soils using the Discrete Element Method**. *The Electronic Journal of Geotechnical Engineering*.

[Tran2012c] Tran, V.D.H., Meguid, M.A., Chouinard, L.E. (2012), **Discrete Element and Experimental Investigations of the Earth Pressure Distribution on Cylindrical Shafts**. *International Journal of Geomechanics*. DOI 10.1061/(ASCE)GM.1943-5622.0000277

[Tran2013] Tran, V.D.H., Meguid, M.A., Chouinard, L.E. (2013), **A finite–discrete element framework for the 3D modeling of geogrid–soil interaction under pullout loading conditions**. *Geotextiles and Geomembranes* (37), pages 1–9. DOI 10.1016/j.geotexmem.2013.01.003

[Tran2011] Tran, V.T., Donzé, F.V., Marin, P. (2011), **A discrete element model of concrete under high triaxial loading**. *Cement and Concrete Composites*.

[Tran2014] Tran, VDH, Meguid, MA, Chouinard, LE (2014), **Three-Dimensional Analysis of Geogrid-Reinforced Soil Using a Finite-Discrete Element Framework**. *International Journal of Geomechanics*.

[Vaddy2022] Vaddy, Poornachandra, Pandurangan, Venkataraman, Biligiri, Krishna Prapoorna (2022), **Discrete element method to investigate flexural strength of pervious concrete**. *Construction and Building Materials* (323), pages 126477.

[Valera2015] Valera, Roberto Rosello, Morales, Irvin Perez, Vanmaercke, Simon, Morfa, Carlos Recarey, Cortes, Lucia Arguelles, Casanas, Harold Diaz-Guzman (2015), **Modified algorithm for generating high volume fraction sphere packings**. *Computational Particle Mechanics*, pages 1–12. DOI 10.1007/s40571-015-0045-8

[vanderHaven2023] van der Haven, Dingeman L. H., Fragkopoulos, Ioannis S., Elliott, James A. (2023), **A physically consistent Discrete Element Method for arbitrary shapes using Volume-interacting Level Sets**. *Computer Methods in Applied Mechanics and Engineering* (414), pages 116165. DOI 10.1016/j.cma.2023.116165

[vanderHaven2024] van der Haven, Dingeman LH, Fragkopoulos, Ioannis S, Elliott, James A (2024), **Volume-interacting level set discrete element method: The porosity and angle of**

repose of aspherical, angular, and concave particles. *Powder Technology* (433), pages 119295.

[vanderLinden2016] van der Linden, Joost H., Narsilio, Guillermo A., Tordesillas, Antoinette (2016), **Machine learning framework for analysis of transport through complex networks in porous, granular media: A focus on permeability**. *Phys. Rev. E* 2 (94), pages 022904. DOI 10.1103/PhysRevE.94.022904

[Varela2023] Varela-Rosales, Nydia Roxana, Santarossa, Angel, Engel, Michael, P"oschel, Thorsten (2023), **Granular binary mixtures improve energy dissipation efficiency of granular dampers**. *Granular Matter* (25), pages 49.

[Verma2025] Verma, Harshal, Mishra, Partha Narayan, Manna, Bappaditya, Williams, David (2025), **Backfill-geogrid interaction: insights from pullout tests and numerical simulation**. *Acta Geotechnica*, pages 1–22.

[Volcy2025] Volcy, Sebastien HE, Sibille, Luc, Chareyre, Bruno, Dano, Christophe, Hosseini-Sadrabadi, Hamid (2025), **An adaptative discretization to model boundary value problems with discrete element method**. *Granular Matter* (27), pages 1–12.

[Wan2014] Wan, R, Khosravani, S, Pouragha, M (2014), **Micromechanical analysis of force transport in wet granular soils**. *Vadose Zone Journal* (13). DOI 10.2136/vzj2013.06.0113

[Wan2015] Wan R., J. Duriez, F. Darve (2015), **A tensorial description of stresses in triphasic granular materials with interfaces**. *Geomechanics for Energy and the Environment* (4), pages 73–87. DOI 10.1016/j.gete.2015.11.004

[Wang2014] Wang, XiaoLiang, Li, JiaChun (2014), **Simulation of triaxial response of granular materials by modified DEM**. *Science China Physics, Mechanics & Astronomy* (57), pages 2297–2308.

[Wang2021] Wang, Tao, Wautier, Antoine, Liu, Sihong, Nicot, Franccois (2021), **How fines content affects granular plasticity of under-filled binary mixtures**. *Acta Geotechnica*, pages 1–15.

[Wang2022] Wang, Xiaoliang, Li, Ge, Liu, Qingquan (2022), **An updated critical state model by incorporating inertial effects for granular material in solid–fluid transition regime**. *Granular Matter* (24), pages 1–9.

[WangYu2023] Wang, Yu, Nie, Jia-Yan, Zhao, Shiwei, Wang, Hao (2023), **A coupled FEM-DEM study on mechanical behaviors of granular soils considering particle breakage**. *Computers and Geotechnics* (160), pages 105529.

[WangTao2023] Wang, Tao, Wautier, Antoine, Zhu, Jungao, Nicot, Franccois (2023), **Stabilizing role of coarse grains in cohesionless overfilled binary mixtures: A DEM investigation**. *Computers and Geotechnics* (162), pages 105625.

[WangJiannan2023] Wang, Jiannan, Uhlemann, Sebastian, Otto, Shawn, Dozier, Brian, Kuhlman, Kristopher L, Wu, Yuxin (2023), **Joint Geophysical and Numerical Insights of the Coupled Thermal-Hydro-Mechanical Processes During Heating in Salt**. *Journal of Geophysical Research: Solid Earth* (128), pages e2023JB026954.

[WangHailin2023] Wang, Hailin, Sun, Hong, Ge, Xiurun, Niu, Fujun (2023), **Macro-micro Performances of Granular Materials Considering the Influences of Density and Stress Path under True Triaxial Conditions: A DEM Investigation**. *KSCE Journal of Civil Engineering* (27), pages 4176–4191.

[Wang2025a] Wang, Ya-Qiong, Feng, Shi-Jin (2025), **Dynamic Responses Generated by Moving Vehicles in a Reinforced Embankment Overlying a Void: Insights from a DEM Study**. *Journal of Geotechnical and Geoenvironmental Engineering* (151), pages 04025059.

[Wang2025b] Wang, Bo (2025), **DEM simulation of the single particle crushing test using an improved fragment replacement method**. *Computational Particle Mechanics* (13), pages 65-78.

[Wang2025c] Wang, Tao, Tang, Chao-Sheng, Lin, Luan, Chen, Ting-Ting, Cai, Zhao-Lin (2025), **Soil desiccation cracking under a non-uniform temperature field: experimental investigation and DEM modeling**. *Journal of Rock Mechanics and Geotechnical Engineering*.

[Wei2022] Wei, Jiangtao (2022), **Particle-void fabric and effective stress reduction in cyclic liquefaction of granular soils**. *Soil Dynamics and Earthquake Engineering* (153), pages 107081.

[Wei2024] Wei, Jiangtao, Xu, Tiejie, He, Jianxian (2024), **Effect of static shear stress and cyclic loading direction on cyclic behaviors of granular soils by DEM analysis**. *Computers and Geotechnics* (167), pages 106112.

[Widulinski2011] Widuliński, J. Kozicki, J. Tejchman, D. Leśniewska (2011), **Discrete simulations of shear zone patterning in sand in earth pressure problems of a retaining wall.**. *International Journal of Solids and Structures* (48), pages 1191–1209. DOI 10.1016/j.ijsolstr.2011.01.005

[Wu2023] Wu, Huanran, Wu, Wei, Liang, Weijian, Dai, Feng, Liu, Hanlong, Xiao, Yang (2023), **3D DEM modeling of biocemented sand with fines as cementing agents**. *International Journal for Numerical and Analytical Methods in Geomechanics* (47), pages 212–240.

[Wu2025] Wu, Fanyu, Sac-Morane, Alexandre, Rattez, Hadrien, Veveakis, Manolis, Hu, Manman (2025), **Onset of reactive brittle cracking in sandstones: DEM-informed phase-field modeling**. *International Journal of Rock Mechanics and Mining Sciences* (196), pages 106319.

[Xiao2023] Xiao, Junhua, Xue, Lihua, Zhang, De, Sun, Siqi, Bai, Yingqi, Shi, Jin (2023), **Coupled DEM-FEM methods for analyzing contact stress between railway ballast and subgrade considering real particle shape characteristic**. *Computers and Geotechnics* (155), pages 105192.

[Xiong2021] Xiong, Hao, Yin, Zhen-Yu, Zhao, Jidong, Yang, Yi (2021), **Investigating the effect of flow direction on suffusion and its impacts on gap-graded granular soils**. *Acta Geotechnica* (16), pages 399–419.

[Xu2022] Xu, Wen-Jie, Wang, Lin, Cheng, Kai (2022), **The Failure and River Blocking Mechanism of Large-Scale Anti-dip Rock Landslide Induced by Earthquake**. *Rock Mechanics and Rock Engineering*, pages 1–21.

[Xuan2025] Xuan, Shenyu, Zhan, Chengsheng, Liu, Zuyuan (2025), **A multi-bonding and damage model for spherical discrete element method**. *Engineering Analysis with Boundary Elements* (179), pages 106387.

[Yang2022] Yang, Siyuan, Huang, Duruo, Wang, Gang, Jin, Feng (2022), **Probing Fabric Evolution and Reliquefaction Resistance of Sands Using Discrete-Element Modeling**. *Journal of Engineering Mechanics* (148), pages 04022023.

[Yang2023a] Yang, Siyuan, Huang, Duruo (2023), **Understanding fabric evolution and multiple liquefaction resistance of sands in the presence of initial static shear stress**. *Soil Dynamics and Earthquake Engineering* (171), pages 107962.

[Yang2023b] Yang, Siyuan, Huang, Duruo (2023), **Investigating the influence of inherent soil fabrics on reliquefaction resistance of sands using DEM-clump simulation**. *Computers and Geotechnics* (164), pages 105817.

[Yang2024] Yang, Siyuan, Huang, Duruo (2024), **Understanding the influence of drained cyclic preloading on liquefaction resistance of sands using DEM-clump modeling**. *Computers and Geotechnics* (176), pages 106800.

[Yim2022] Yim, Seungkyun, Bian, Huakang, Aoyagi, Kenta, Yamanaka, Kenta, Chiba, Akihiko (2022), **Spreading behavior of Ti48Al2Cr2Nb powders in powder bed fusion additive manufacturing process: Experimental and discrete element method study**. *Additive Manufacturing* (49), pages 102489.

[Yim2023] Yim, Seungkyun, Aoyagi, Kenta, Yanagihara, Keiji, Bian, Huakang, Chiba, Akihiko (2023), **Effect of mechanical ball milling on the electrical and powder bed properties of**

gas-atomized Ti–48Al–2Cr–2Nb and elucidation of the smoke mechanism in the powder bed fusion electron beam melting process. *Journal of Materials Science & Technology* (137), pages 36–55.

[Yim2025] Yim, Seungkyun, Wang, Hao, Aoyagi, Kenta, Yamanaka, Kenta, Chiba, Akihiko (2025), **Comparative evaluation of powder spreading strategies to enhance powder bed quality in powder bed fusion additive manufacturing: A DEM simulation study**. *Powder Technology* (453), pages 120614.

[Yousefpour2022] Yousefpour, Negin, Pouragha, Mehdi (2022), **Prediction of the post-failure behavior of rocks: Combining artificial intelligence and acoustic emission sensing**. *International Journal for Numerical and Analytical Methods in Geomechanics*.

[Yuan2016] Yuan C., B. Chareyre, F. Darve (2016), **Pore-scale simulations of drainage in granular materials: finite size effects and the representative elementary volume**. *Adv. in Water Ressources* (95), pages 109–124.

[Yuan2017] Yuan C., B. Chareyre (2017), **A pore-scale method for hydromechanical coupling in deformable granular media**. *Computer Methods in Applied Mechanics and Engineering*. DOI 10.1016/j.cma.2017.02.024

[Yuan2018] Yuan, Chao, Chareyre, Bruno, Darve, Félix (2018), **Deformation and stresses upon drainage of an idealized granular material**. *Acta Geotechnica* (13). DOI 10.1016/j.cma.2017.02.024

[Yuan2019] Yuan, Chao, Moscariello, Mariagiovanna, Cuomo, Sabatino, Chareyre, Bruno (2019), **Numerical simulation of wetting-induced collapse in partially saturated granular soils**. *Granular Matter* (21), pages 64.

[Zang2024] Zang, Haizhi, Wang, Shanyong, Carter, John P (2024), **Analysis of thixotropy of cement grout based on a virtual bond model**. *Acta Geotechnica* (19), pages 7427–7450.

[Zhang2021] Zhang, Lingran, Scholtès, Luc, Donzé, Frédéric Victor (2021), **Discrete Element Modeling of Permeability Evolution During Progressive Failure of a Low-Permeable Rock Under Triaxial Compression**. *Rock Mechanis and Rock Engineering*. DOI 10.1007/s00603-021-02622-9

[Zhang2023] Zhang, Aoxi, Dieudonné, Anne-Catherine (2023), **Effects of carbonate distribution pattern on the mechanical behaviour of bio-cemented sands: A DEM study**. *Computers and Geotechnics* (154), pages 105152.

[Zhang2025a] Zhang, Chao, O'Shaughnessy, Connor, Maramizonouz, Sadaf, Angelidakis, Vasileios, Nadimi, Sadegh (2025), **Controlling fragment size distribution for modelling the breakage of multi-sphere particles**. *Particuology* (98), pages 105–116.

[Zhang2025b] Zhang, Aoxi, Wang, Liang, Zhang, Wengang, Zhao, Chaofa, Zhang, Pan (2025), **Bayesian Neural Network Prediction and Uncertainty Analysis of Bio-Cemented Soil Strength**. *International Journal for Numerical and Analytical Methods in Geomechanics*.

[ZhangAoxi2024] Zhang, Aoxi, Dieudonné, Anne-Catherine (2024), **Cementor: A toolbox to generate bio-cemented soils with specific microstructures**. *Biogeotechnics*, pages 100081.

[ZhangShihao2024] Zhang, Shihao, Zhang, Yingbin, Wei, Jiangtao, Jiang, Minxuan, Qin, Yiqiao (2024), **Effects of initial static shear on liquefaction behaviour of Toyoura sand in undrained multi direction dynamic cyclic simple shear tests and DEM**. *Japanese Geotechnical Society Special Publication* (10), pages 1521–1526.

[Zhao2015] Zhao J., N. Guo (2015), **The interplay between anisotropy and strain localisation in granular soils: a multiscale insight**. *Géotechnique*. (under review)

[Zhao2017] Zhao, Shiwei, Zhang, Nan, Zhou, Xiaowen, Zhang, Lei (2017), **Particle shape effects on fabric of granular random packing**. *Powder Technology* (310), pages 175–186.

[Zhao2019] Zhao, Benzhong, MacMinn, Christopher W, Primkulov, Bauyrzhan K, Chen, Yu, Valocchi, Albert J, Zhao, Jianlin, Kang, Qinjun, Bruning, Kelsey, McClure, James E, Miller, Cass T, others (2019), **Comprehensive comparison of pore-scale models for multiphase**

flow in porous media. *Proceedings of the National Academy of Sciences* (116), pages 13799–13806.

[Zhao2021a] Zhao, Shiwei, Zhao, Jidong (2021), **SudoDEM: Unleashing the predictive power of the discrete element method on simulation for non-spherical granular particles**. *Computer Physics Communications* (259), pages 107670.

[Zhao2021b] Zhao, Yufan, Koizumi, Yuichiro, Aoyagi, Kenta, Yamanaka, Kenta, Chiba, Akihiko (2021), **Thermal properties of powder beds in energy absorption and heat transfer during additive manufacturing with electron beam**. *Powder Technology* (381), pages 44–54.

[Zhao2022] Zhao, Chaofa, Kruyt, Niels P, Pouragha, Mehdi, Wan, Richard (2022), **Fabric response to stress probing in granular materials: Two-dimensional, anisotropic systems**. *Computers and Geotechnics* (146), pages 104695.

[Zhao2023a] Zhao, Yufan, Aoyagi, Kenta, Cui, Yujie, Yamanaka, Kenta, Chiba, Akihiko (2023), **Multiscale heat transfer affected by powder characteristics during electron beam powder-bed fusion**. *Powder Technology* (421), pages 118438.

[Zhao2023b] Zhao, Yufan, Aoyagi, Kenta, Yamanaka, Kenta, Chiba, Akihiko (2023), **Processing condition dependency of increased layer thickness on surface quality during electron beam powder bed fusion**. *Journal of Materials Research and Technology* (26), pages 5264–5279.

[Zhao2025] Zhao, Chaofa, Duan, Ge, Yang, Zhongxuan (2025), **Three-dimensional micromechanical expression for the average strain tensor of granular materials**. *Journal of the Mechanics and Physics of Solids*, pages 106189.

[Zhi2024] Zhi, Peng, Wu, Yu-Ching, Bai, Meiyan (2024), **Determining the effect of geometric and dynamic properties of screws on fiber orientation during FRC 3D printing based on discrete element simulation**. *Automation in Construction* (165), pages 105513.

[Zhong2021] Zhong, Xinran, Sun, WaiChing, Dai, Ying (2021), **A reduced-dimensional explicit discrete element solver for simulating granular mixing problems**. *Granular Matter* (23), pages 1–13.

[Zhuo2025] Zhuo, Longchao, Jiang, Chenghao, Xu, Jintao, Wang, Hao, Huang, Bin, Zhang, Danli, Wang, Yanlin (2025), **Microstructural evolution and performance optimization of SLM-fabricated W/AlSi10Mg composites**. *Journal of Alloys and Compounds* (1010), pages 178317.

[1stYadeWorkshop] B. Chareyre (ed.), **Booklet of presentations of the 1st Yade Workshop** (2014).

[2ndYadeWorkshop] J. Duriez (ed.), **2nd YADE Workshop: Discrete-based modeling of multi-scale coupled problems. Booklet of Abstracts** (2018).

[Aboul2016b] Aboul Hosn, R, Sibille, L, Benahmed, N, Chareyre, B (2016), **A discrete numerical description of the mechanical response of soils subjected to degradation by suffusion**. In *Scour and Erosion: Proceedings of the 8th International Conference on Scour and Erosion (Oxford, UK, 12-15 September 2016)*.

[Aboul2017b] Aboul Hosn, Rodaina, Nguyen, Cong Doan, Sibille, Luc, Benahmed, Nadia, Chareyre, Bruno (2017), **Microscale Analysis of the Effect of Suffusion on Soil Mechanical Properties**. In *International Workshop on Bifurcation and Degradation in Geomaterials*.

[Albaba2015b] Albaba, Adel, Lambert, Stéphane, Nicot, François, Chareyre, Bruno (2015), **Modeling the Impact of Granular Flow against an Obstacle**. In *Recent Advances in Modeling Landslides and Debris Flows*.

[Angelidakis2025a] Angelidakis, Vasileios, Nadimi, Sadegh, Utili, Stefano (2025), **A particle shape classification strategy to inform the generation of representative element volumes in the discrete element method**. In *IOP Conference Series: Earth and Environmental Science*.

[Angelidakis2025b] Angelidakis, Vasileios, Duverger, Sacha, Nadimi, Sadegh, Utili, Stefano, Bonelli, St'ephane, Philippe, Pierre, Duriez, J'er^ome (2025), **Measuring and predicting the**

angle of repose of granular matter from clump and potential particles DEM approaches. In *IOP Conference Series: Earth and Environmental Science*.

[Bhatpahri2025] Bhatpahri, Pramay, Gopalakrishnan, Srinivasan (2025), **Prediction of Small-Strain Properties of Dry Sand Using Curve Fitting and Machine Learning Models**. In *EPJ Web of Conferences*.

[Bonilla2014] Bonilla-Sierra, V, Donzé, FV, Scholtès, L, Elmouttie, M (2014), **Coupling photogrammetric data with a discrete element model for rock slope stability assessment**. In *Rock Engineering and Rock Mechanics: Structures in and on Rock Masses*.

[Bourrier2015b] Bourrier, Franck, Baroth, Julien, Lambert, Stéphane (2015), **How Can Reliability-Based Approaches Improve the Design of Rockfall Protection Fences?**. In *Engineering Geology for Society and Territory-Volume 2*.

[Catalano2011a] Catalano E., B. Chareyre, A. Cortis, E. Barthélémy (2011), **A Pore-Scale Hydro-Mechanical coupled model for geomaterials**. In *II International Conference on Particle-based Methods - Fundamentals and Applications*.

[Catalano2010b] Catalano E., B. Chareyre, E. Barthélémy (2010), **Pore scale modelling of Stokes flow**. In *GdR MeGe*.

[Catalano2013b] Catalano E., Chareyre B., Barthélémy E. (2013), **DEM-PFV analysis of solid-fluid transition in granular sediments under the action of waves**. In *Powders and Grains 2013: Proceedings of the 6th International Conference on Micromechanics of Granular Media. AIP Conference Proceedings*. DOI 10.1063/1.4812118

[Catalano2009] Catalano E., B. Chareyre, E. Barthélémy (2009), **Fluid-solid coupling in discrete models**. In *Alert Geomaterials Workshop 2009*.

[Catalano2010a] Catalano E., B. Chareyre, E. Barthélémy (2010), **A coupled model for fluid-solid interaction analysis in geomaterials**. In *Alert Geomaterials Workshop 2010*.

[Caulk2019a] Caulk, R.A., Chareyre, B. (2019), **An open framework for the simulation of coupled Thermo-Hydro-Mechanical processes in Discrete Element Systems**. In *8th International Conference on Discrete Element Methods*.

[Caulk2019b] Caulk, R.A.„ Kozicki, J., Kunhappan, D.„ Maurin, R.„ Montellà, E.P.„ Sweijen, T.„ Yuan, C.„ Chareyre, B. (2019), **Yade's (undeniable) transformation into a model project of optimzation, multi-physics couplings, and user support**. In *8th International Conference on Discrete Element Methods*.

[Caulk2019c] Caulk, Robert A, Kozicki, Janek, Kunhappan, Deepak, Maurin, Rapha"el (2019), **Yade's (undeniable) transformation into a model project of optimization, multi-physics couplings, and user support**. In *8th International Conference on Discrete Element Methods (DEM8)*.

[Chareyre2011] Chareyre B., E. Catalano, E. Barthélémy (2011), **Numerical simulation of hydromechanical couplings by combined discrete element method and finite-volumes**. In *International Conference on Flows and Mechanics in Natural Porous Media from Pore to Field Scale - Pore2Field*.

[Chareyre2017b] Chareyre B., E. Nikooee , C. Chalak , C. Yuan (2017), **Micromechanical Insights into the Effective Stresses**. In *Poromechanics VI*. DOI 10.1061/9780784480779.051

[Chareyre2012b] Chareyre B., L. Scholtès, F. Darve (2012), **The properties of some stress tensors investigated by DEM simulations**. In *Euromech Colloquium 539; Mechanics of Unsaturated Porous Media: Effective Stress principle; from micromechanics to thermodynamics*

[Chareyre2009] Chareyre B., Scholtès L. (2009), **Micro-statics and micro-kinematics of capillary phenomena in dense granular materials**. In *POWDERS AND GRAINS 2009: Proceedings of the 6th International Conference on Micromechanics of Granular Media. AIP Conference Proceedings*. DOI 10.1063/1.3180083

[Chareyre2019] Chareyre B.„ Montellà, E.P.„ Yuan, C.„ Gens, A. (2019), **A hybrid pore network - LBM method for integrating flow of immiscible phases in DEM**. In *8th International Conference on Discrete Element Methods*.

[Chareyre2012c] Chareyre B. (2012), **Micro-poromechanics: recent advances in numerical models and perspectives**. In *ICACM symposium 2012, The role of structure on emerging material properties*

[Chareyre2017] Chareyre, Bruno, Yuan, Chao, Montella, Eduard P, Salager, Simon (2017), **Toward multiscale modelings of grain-fluid systems**. In *EPJ Web of Conferences*.

[Chen2008a] Chen, F., Drumm, E.C., Guiochon, G., Suzuki, K. (2008), **Discrete Element Simulation of 1D Upward Seepage Flow with Particle-Fluid Interaction Using Coupled Open Source Software**. In *Proceedings of The 12th International Conference of the International Association for Computer Methods and Advances in Geomechanics (IACMAG) Goa, India*.

[Chen2009b] Chen, F., Drumm, E.C., Guiochon, G. (2009), **3D DEM Analysis Of Graded Rock Fill Sinkhole Repair: Particle Size Effects On The Probability Of Stability**. In *Transportation Research Board Conference (Washington DC)*.

[Dang2008a] Dang, H.K., Mohamed, M.A. (2008), **An algorithm to generate a specimen for Discrete Element simulations with a predefined grain size distribution.**. In *61th Canadian Geotechnical Conference, Edmonton, Alberta*.

[Dang2008b] Dang, H.K., Mohamed, M.A. (2008), **3D simulation of the trap door problem using the Discrete Element Method.**. In *61th Canadian Geotechnical Conference, Edmonton, Alberta*.

[Darve2016] Darve F., J. Duriez, R. Wan (2016), **DEM modelling in Geomechanics: some recent breakthroughs**. In *Proceedings of the 7th International Conference on Discrete Element Methods*.

[delValle2025] del Valle, Carlos Andr'es, Angelidakis, Vasileios, Roy, Sudeshna, Mu~noz, Jos'e Daniel, P"oschel, Thorsten (2025), **Efficient numerical integration of rigid body dynamics**. In *EPJ Web of Conferences*.

[Duriez2020] Duriez J., C. Galusinski (2020), **Level Set Representation on Octree for Granular Material with Arbitrary Grain Shape**. In *Proceedings Topical Problems of Fluid Mechanics 2020*. DOI 10.14311/TPFM.2020.009

[Duriez2017e] Duriez J., R. Wan, F. Darve (2017), **Microstructural Views of Stresses in Three-Phase Granular Materials**. In *From Microstructure Investigations to Multiscale Modeling: Bridging the Gap* (D. Brancherie, P. Feissel, S. Bouvier and A. Ibrahimbegovic, ed.), John Wiley & Sons, Inc. , DOI 10.1002/9781119476757.ch6

[Duriez2017d] Duriez J., R. Wan, M. Pouragha (2017), **Partially Saturated Granular Materials: Insights From Micro-Mechanical Modelling**. In *6th Biot Conference on Poromechanics*.

[Duriez2015] Duriez J., R. Wan (2015), **Effective stress in unsaturated granular materials: micro-mechanical insights**. In *Coupled Problems in Science and Engineering VI*.

[Duriez2019] Duriez J., S. Duverger, R. Wan (2019), **A micromechanical, UNSAT, approach for wet granular soils**. In *8th International Conference on Discrete Element Methods*.

[Duriez2022a] Duriez, J., Galusinski, C., Golay, F., Bonelli, S. (2022), **Modélisations numériques discrètes de matériaux granulaires à partir d'une description Level Set des particules**. In *25e Congr'es Franccais de M'ecanique (CFM 2022)*.

[Duriez2022b] Duriez, J., Galusinski, C., Golay, F., Bonelli, S. (2022), **A discrete element method for granular solids with a level set shape description**. In *The 8th European Congress on Computational Methods in Applied Sciences and Engineering (ECCOMAS Congress 2022)*.

[Duriez2023] Duriez, J., Golay, F., Bonelli, S., Galusinski, C. (2023), **A Level Set approach for non-spherical DEM in YADE**. In *DEM9: 9th International Conference on Discrete Element Methods*.

[Effeindzourou2015] Effeindzourou A., K. Thoeni, A. Giacomini, S.W. Sloan (2015), **A discrete model for rock impacts on muckpiles**. In *Computer Methods and Recent Advances in Geomechanics*.

[Effeindzourou2015a] Effeindzourou, A., Thoeni, K., Chareyre, B., Giacomini, A. (2015), **A general method for modelling deformable structures in DEM**. In *Particle-Based Methods IV: Fundamentals and Applications*.

[Elias2013] Eliáš, J. (2013), **Dem simulation of railway ballast using polyhedral elemental shapes**. In *Particle-Based Methods III: Fundamentals and Applications - Proceedings of the 3rd International Conference on Particle-based MethodsFundamentals and Applications, Particles 2013*.

[Farahnak2022] Farahnak, Mojtaba, Wan, Richard, Pouragha, Mehdi, Nicot, Franccois (2022), **Perturbations in Granular Materials: Subtleties in DEM Modeling**. In *International Workshop on Bifurcation and Degradation in Geomaterials*.

[Favier2009b] Favier L., D. Daudon, F. Donzé, J. Mazars (2009), **Validation of a DEM granular flow model aimed at forecasting snow avalanche pressure**. In *AIP Conference Proceedings*. DOI 10.1063/1.3180002

[Frey2017] Frey P., R. Maurin, L. Morchid-Alaoui, S. Gupta, J. Chauchat (2017), **Vertical size segregation numerical experiments in bedload sediment transport with a coupled fluid-discrete element model**. In *Proceedings of Powder and Grains 2017*.

[Gillibert2009] Gillibert L., Flin F., Rolland du Roscoat S., Chareyre B., Philip A., Lesaffre B., Meyssonier J. (2009), **Curvature-driven grain segmentation: application to snow images from X-ray microtomography**. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Miami, USA)*.

[Gladky2015a] Gladky, Anton, Lieberwirth, Holger, Schwarze, Ruediger (2015), **DEM simulation of the dry and weakly wetted bulk flow on a pelletizing table**. In *13th U.S. National Congress on Computational Mechanics*.

[Gladky2015b] Gladky, Anton, Roy, Sudeshna, Weinhart, Thomas, Luding, Stefan, Schwarze, Ruediger (2015), **DEM simulations of weakly wetted granular materials: implementation of capillary bridge models**. In *Fourth Conference on Particle-Based Methods (PARTICLES 2015)*.

[Guo2013] Guo Ning, Jidong Zhao (2013), **A hierarchical model for cross-scale simulation of granular media**. In *Powders and Grains 2013: Proceedings of the 6th International Conference on Micromechanics of Granular Media. AIP Conference Proceedings*. DOI 10.1063/1.4812158

[Guo2014b] Guo, Ning, Zhao, Jidong (2015), **A Multiscale Investigation of Strain Localization in Cohesionless Sand**. In *Bifurcation and Degradation of Geomaterials in the New Millennium* (Chau, Kam-Tim and Zhao, Jidong, ed.), DOI 10.1007/978-3-319-13506-9_18

[Hadda2013b] Hadda Nejib, François Nicot, Luc Sibille, Farhang Radjai, Antoinette Tordesillas, Félix Darve (2013), **A multiscale description of failure in granular materials**. In *Powders and Grains 2013: Proceedings of the 6th International Conference on Micromechanics of Granular Media. AIP Conference Proceedings*. DOI 10.1063/1.4811999

[Hadda2015b] Hadda, N, Bourrier, F, Sibille, L, Nicot, F, Wan, R, Darve, F (2015), **A microstructural cluster-based description of diffuse and localized failures**. In *Geomechanics from Micro to Macro: Proceedings of the International Symposium on Geomechanics from Micro and Macro (IS-Cambridge 2014)*.

[Harthong2013] Harthong Barthélémy, Richard G Wan (2009), **Directional plastic flow and fabric dependencies in granular materials**. In *Powders and Grains 2013: Proceedings of the 6th International Conference on Micromechanics of Granular Media. AIP Conference Proceedings*. DOI http://dx.doi.org/10.1063/1.4811900

[Hasan2010b] Hasan A., B. Chareyre, J. Kozicki, F. Flin, F. Darve, J. Meyssonnier (2010), **Microtomography-based Discrete Element Modeling to Simulate Snow Microstructure Deformation**. In *AGU Fall Meeting Abstracts*

[Hicher2009] Hicher P.-Y., Scholtès L., Chareyre B., Nicot F., Darve F. (2008), **On the capillary stress tensor in wet granular materials**. In *Inaugural International Conference of the Engineering Mechanics Institute (EM08) - (Minneapolis, USA)*.

[Hicher2011] Hicher, P.Y, Scholtès, L., Sibille, L. (2011), **Multiscale Modeling of Particle Removal Impact on Granular Material Behavior**. In *Engineering Mechanics Institute, EMI 2011*.

[Hilton2013b] Hilton J. E., P. W. Cleary, A. Tordesillas (2013), **Unitary stick-slip motion in granular beds**. In *Powders and Grains 2013: Proceedings of the 6th International Conference on Micromechanics of Granular Media. AIP Conference Proceedings*. DOI 10.1063/1.4812063

[Huang2022] Huang, Duruo, Yuan, Zhengxin, Yang, Siyuan, Amini, Pedram Fardad, Wang, Gang, Jin, Feng (2022), **Multiple Liquefaction of Granular Soils: A New Stacked Ring Torsional Shear Apparatus and Discrete Element Modeling**. In *Conference on Performance-based Design in Earthquake. Geotechnical Engineering*.

[Jahanshahinowkandeh2025] Jahanshahinowkandeh, M, Barreto, D, Miranda, M, Rodriguez-Fernandez, E, Castro, J (2025), **Skirt penetration in scour protection: laboratory and numerical analyses**. In *Proceedings of ISFOG 2025*.

[Kajiyama2025] Kajiyama, S, Nakata, Y (2025), **Fundamental study on the relationship between the apex and internal immobile area during sand heap formation**. In *IOP Conference Series: Earth and Environmental Science*.

[Kalogeropoulos2023] Kalogeropoulos, A, Michalakopoulos, T (2023), **Numerical simulation of the rock cutting process using the discrete element method**. In *Expanding Underground-Knowledge and Passion to Make a Positive Impact on the World* CRC Press ,

[Kozicki2007c] Kozicki J., J. Tejchman (2007), **Simulations of fracture processes in concrete using a 3D lattice model**. In *Int. Conf. on Computational Fracture and Failure of Materials and Structures (CFRAC 2007), Nantes*.

[Kozicki2007d] Kozicki J., J. Tejchman (2007), **Effect of aggregate density on fracture process in concrete using 2D discrete lattice model**. In *Proc. Conf. Computer Methods in Mechanics (CMM 2007), Lodz-Spala*.

[Kozicki2007e] Kozicki J., J. Tejchman (2007), **Modelling of a direct shear test in granular bodies with a continuum and a discrete approach**. In *Proc. Conf. Computer Methods in Mechanics (CMM 2007), Lodz-Spala*.

[Kozicki2007f] Kozicki J., J. Tejchman (2007), **Investigations of size effect in tensile fracture of concrete using a lattice model**. In *Proc. Conf. Modelling of Heterogeneous Materials with Applications in Construction and Biomedical Engineering (MHM 2007), Prague*.

[Kozicki2006b] Kozicki J., J. Tejchman (2006), **Modelling of fracture process in brittle materials using a lattice model**. In *Computational Modelling of Concrete Structures, EURO-C (eds.: G. Meschke, R. de Borst, H. Mang and N. Bicanic), Taylor anf Francis*.

[Kozicki2006c] Kozicki J., J. Tejchman (2006), **Lattice type fracture model for brittle materials**. In *35th Solid Mechanics Conference (SOLMECH 2006), Krakow*.

[Kozicki2005b] Kozicki J., J. Tejchman (2005), **Simulations of fracture in concrete elements using a discrete lattice model**. In *Proc. Conf. Computer Methods in Mechanics (CMM 2005), Czestochowa, Poland*.

[Kozicki2005c] Kozicki J., J. Tejchman (2005), **Simulation of the crack propagation in concrete with a discrete lattice model**. In *Proc. Conf. Analytical Models and New Concepts in Concrete and Masonry Structures (AMCM 2005), Gliwice, Poland*.

[Kozicki2004a] Kozicki J., J. Tejchman (2004), **Study of Fracture Process in Concrete using a Discrete Lattice Model**. In *CURE Workshop, Simulations in Urban Engineering, Gdansk*.

[Kozicki2003a] Kozicki J., J. Tejchman (2003), **Discrete methods to describe the behaviour of quasi-brittle and granular materials**. In *16th Engineering Mechanics Conference, University of Washington, Seattle, CD–ROM*.

[Kozicki2003c] Kozicki J., J. Tejchman (2003), **Lattice method to describe the behaviour of quasi-brittle materials**. In *CURE Workshop, Effective use of building materials, Sopot*.

[Kozicki2011] Kozicki J., J. Tejchman (2011), **Numerical simulation of sand behaviour using DEM with two different descriptions of grain roughness**. In *II International Conference on Particle-based Methods - Fundamentals and Applications*.

[Kozicki2013] Kozicki Jan, Jacek Tejchman, Danuta Lesniewska (2013), **Study of some micro-structural phenomena in granular shear zones**. In *Powders and Grains 2013: Proceedings of the 6th International Conference on Micromechanics of Granular Media. AIP Conference Proceedings*. DOI 10.1063/1.4811976

[Kozicki2005a] Kozicki, J. (2005), **Discrete lattice model used to describe the fracture process of concrete**. In *Discrete Element Group for Risk Mitigation Annual Report 1, Grenoble University of Joseph Fourier, France*.

[Krzaczek2022a] Krzaczek, M, Nitka, M, Tejchman, J (2022), **Modeling of capillary fluid flow in concrete using a DEM-CFD approach**. In *Computational Modelling of Concrete and Concrete Structures* CRC Press ,

[Krzaczek2022b] Krzaczek, M, Nitka, M, Tejchman, J (2022), **A novel DEM based pore-scale thermo-hydro-mechanical model**. In *Computational Modelling of Concrete and Concrete Structures* CRC Press ,

[Larijani2025] Larijani, Roxana Saghafian, Luding, Stefan, Magnanimo, Vanessa (2025), **Wet granulation multiscale modelling accelerated via CG-DEM**. In *EPJ Web of Conferences*.

[Li2014] Li, W, Vincens, E, Reboul, N, Chareyre, B (2014), **Constrictions and filtration of fine particles in numerical granular filters: Influence of the fabric within the material**. In *Scour and Erosion: Proceedings of the 7th International Conference on Scour and Erosion, Perth, Australia, 2-4 December 2014*.

[Lomine2010b] Lominé, F., Poullain, P., Sibille, L. (2010), **Modelling of fluid-solid interaction in granular media with coupled LB/DE methods: application to solid particle detachment under hydraulic loading**. In *19th Discrete Simulation of Fluid Dynamics, DSFD 2010*.

[Lomine2010a] Lominé, F., Scholtès, L., Poullain, P., Sibille, L. (2010), **Soil microstructure changes induced by internal fluid flow: investigations with coupled DE/LB methods**. In *Proc. of 3rd Euromediterranean Symposium on Advances in Geomaterials and Structures, AGS'10*.

[Lomine2011] Lominé, F., Sibille, L., Marot, D. (2011), **A coupled discrete element - lattice Botzmann method to investigate internal erosion in soil**. In *Proc. 2nd Int. Symposium on Computational Geomechanics (ComGeo II)*.

[Maurin2013] Maurin R., B. Chareyre, J. Chauchat, P. Frey (2013), **Discrete element modelling of bedload transport**. In *Proceedings of THESIS 2013, Two-pHase modElling for Sediment dynamIcS in Geophysical Flows*.

[Maurin2015] Maurin R., B. Chareyre, J. Chauchat, P. Frey (2015), **On granular rheology in bedload transport**. In *CFM-2015*.

[Maurin2016b] Maurin R., J. Chauchat, P. Frey (2016), **Dense granular flow rheology in turbulent bedload transport**. In *Proceedings of THESIS 2017, Two-pHase modElling for Sediment dynamIcS in Geophysical Flows*.

[Michallet2012] Michallet H., E. Catalano, C. Berni, V. Rameliarison, E. Barthélémy (2012), **Physical and numerical modelling of sand liquefaction in waves interacting with a vertical wall**. In *ICSE6 - 6th International conference on Scour and Erosion*

[Modenese2012] Modenese C., S. Utili, G.T. Houlsby (2012), **DEM Modelling of Elastic Adhesive Particles with Application to Lunar Soil**. In *Earth and Space 2012: Engineering, Science, Construction, and Operations in Challenging Environments –© 2012 ASCE*. DOI 10.1061/9780784412190.006

[Modenese2012a] Modenese, C, Utili, S, Houlsby, G T (2012), **A study of the influence of surface energy on the mechanical properties of lunar soil using DEM**. In *Discrete Element Modelling of Particulate Media* (Wu, Chuan-Yu, ed.), Royal Society of Chemistry , DOI 10.1039/9781849735032-00069

[Modenese2012b] Modenese, C, Utili, S, Houlsby, G T (2012), **A numerical investigation of quasi-static conditions for granular media**. In *Discrete Element Modelling of Particulate Media* (Wu, Chuan-Yu, ed.), Royal Society of Chemistry , DOI 10.1039/9781849735032-00187

[Montella2017] Montellà, Eduard Puig, Toraldo, Marcella, Chareyre, Bruno, Sibille, Luc (2017), **From continuum analytical description to discrete numerical modelling of localized fluidization in granular media**. In *EPJ Web of Conferences*.

[Nicot2015] Nicot, F, Hadda, N, Bourrier, F, Sibille, L, Tordesillas, A, Darve, F (2015), **Micromechanical Analysis of Second Order Work in Granular Media**. In *Bifurcation and Degradation of Geomaterials in the New Millennium*.

[Nicot2011b] Nicot, F., Hadda, N., Bourrier, F., Sibille, L., Darve, F. (2011), **A discrete element analysis of collapse mechanisms in granular materials**. In *Proc. 2nd Int. Symposium on Computational Geomechanics (ComGeo II)*.

[Nicot2010] Nicot, F., Sibille, L., Daouadji, A., Hicher, P.Y., Darve, F. (2010), **Multiscale modeling of instability in granular materials**. In *Engineering Mechanics Institute, EMI 2010*.

[Nikooee2017] Nikooee, E, Habibagahi, G, Daneshian, B, Sweijen, T, Hassanizadeh, SM (2017), **A grain scale model to predict retention properties of unsaturated soils**. In *Proceedings of the 19th International COnference on Soil Mechanics and Geotechnical Engineering*.

[Nikooee2016] Nikooee, E, Sweijen, T, Hassanizadeh, SM (2016), **Determination of the relationship among capillary pressure, saturation and interfacial area: a pore unit assembly approach**. In *E3S Web of Conferences*. DOI 10.1051/e3sconf/20160902002

[Nitka2014c] Nitka, M, Tejchman, J, Kozicki, J, Lesniewska, D (2014), **Effect of mean grain diameter on vortices, force chains and local volume changes in granular shear zones**. In *Digital Humanitarians: How Big Data Is Changing the Face of Humanitarian Response*.

[Nitka2014] Nitka, M, Tejchman, J (2014), **Discrete modeling of micro-structure evolution during concrete fracture using DEM**. In *Computational Modelling of Concrete Structures*.

[Nitka2014b] Nitka, M, Tejchman, J (2014), **Discrete modeling of micro-structure evolution during concrete fracture using DEM**. In *Computational Modelling of Concrete Structures*.

[Nitka2015c] Nitka, Michal, Tejchman, Jacek, Kozicki, Jan (2015), **Discrete Modelling of Micro-structural Phenomena in Granular Shear Zones**. In *Bifurcation and Degradation of Geomaterials in the New Millennium*.

[Pramay2025] Pramay, Bhatpahri, Gopalakrishnan, S, Shembekar, Anay Mohan (2025), **Constitutive Modelling of Granular Materials Using Cundall-Strack and Hertz-Mindlin Contact Models**. In *ASME Aerospace Structures, Structural Dynamics, and Materials Conference*.

[Romanova2020] Romanova, Daria, Strijhak, Sergei, Kraposhin, Matvey (2020), **Development of snowYadeFoam solver for snow particles simulation**. In *2020 Ivannikov Ispras Open Conference (ISPRAS)*.

[Sari2011] Sari H., B. Chareyre, E. Catalano, P. Philippe, E. Vincens (2011), **Investigation of Internal Erosion Processes using a Coupled DEM-Fluid Method**. In *II International Conference on Particle-based Methods - Fundamentals and Applications*.

[Sayeed2014] Sayeed, Md Abu, Sazzad, Md Mahmud, Suzuki, Kiichi (2014), **Mechanical Behavior of Granular Materials Considering Confining Pressure Dependency**. In *GeoCongress 2012*.

[Scholtes2009e] Scholtes L, Chareyre B, Darve F (2009), **Micromechanics of partialy saturated granular material**. In *Int. Conf. on Particle Based Methods, ECCOMAS-Particles*.

[Scholtes2008a] Scholtès L., B. Chareyre, F. Nicot, F. Darve (2008), **Capillary Effects Modelling in Unsaturated Granular Materials**. In *8th World Congress on Computational Mechanics - 5th European Congress on Computational Methods in Applied Sciences and Engineering, Venice*.

[Scholtes2007a] Scholtès L., B. Chareyre, F. Nicot, F. Darve (2007), **Micromechanical Modelling of Unsaturated Granular Media**. In *Proceedings ECCOMAS-MHM07, Prague*.

[Scholtes2011b] Scholtès L., F. Donzé (2011), **Progressive failure mechanisms in jointed rock: insight from 3D DEM modelling**. In *II International Conference on Particle-based Methods - Fundamentals and Applications*.

[Scholtes2008b] Scholtès L., P.-Y. Hicher, F.Nicot, B. Chareyre, F. Darve (2008), **On the Capillary Stress Tensor in unsaturated granular materials**. In *EM08: Inaugural International Conference of the Engineering Mechanics Institute, Minneapolis*.

[Scholtes2011c] Scholtès, L., Hicher, P.Y., Sibille, L. (2011), **A micromechanical approach to describe internal erosion effects in soils**. In *Proc. of Geomechanics and Geotechnics: from micro to macro, IS-Shanghai 2011*.

[Shiu2007a] Shiu W., F.V. Donze, L. Daudeville (2007), **Discrete element modelling of missile impacts on a reinforced concrete target**. In *Int. Conf. on Computational Fracture and Failure of Materials and Structures (CFRAC 2007), Nantes*.

[Shoemaker2024] Shoemaker, Travis A, Tanissa, Carine, Hashash, Youssef MA (2024), **Comparison of DEM Software with Polyhedral Particle Shapes**. In *Geo-Congress 2024*.

[Sibille2009] Sibille, L., Scholtès, L. (2009), **Effects of internal erosion on mechanical behaviour of soils: a dem approach**. In *Proc. of International Conference on Particle-Based Methods, Particles 2009*.

[Skarzynski2014] Skarżynski, M. Nitka, J. Tejchman (2014), **Two-scale model for concrete beams subjected to three point bending—numerical analyses and experiments**. In *Computational Modelling of Concrete Structures*.

[Smilauer2007a] Šmilauer V. (2007), **Discrete and hybrid models: Applications to concrete damage**. In *Unpublished*.

[Smilauer2008] Šmilauer Václav (2008), **Commanding c++ with Python**. In *ALERT Doctoral school talk*.

[Smilauer2010a] Šmilauer Václav (2010), **Yade: past, present, future**. In *Internal seminary in Laboratoire 3S-R, Grenoble*. (LaTeX sources)

[Stransky2011] Stransky J., M. Jirasek (2011), **Calibration of particle-based models using cells with periodic boundary conditions**. In *II International Conference on Particle-based Methods - Fundamentals and Applications*.

[Stransky2015a] Stránský Jan, Martin Doškář (2015), **Comparison of DEM-based Wang tilings and PUC**. In *Nano and Macro Mechanics 2015*.

[Stransky2015b] Stránský Jan, Martin Doškář (2015), **Stochastic Wang tiles generation using the discrete element method**. In *Engineering Mechanics 2015*.

[Stransky2010] Stránský Jan, Milan Jirásek, Václav Šmilauer (2010), **Macroscopic elastic properties of particle models**. In *Proceedings of the International Conference on Modelling and Simulation 2010, Prague*.

[Stransky2010b] Stránský Jan, Milan Jirásek, Václav Šmilauer (2010), **Macroscopic Properties of Particle Models: Uniaxial Tension**. In *Nano and Macro Mechanics 2010*.

[Stransky2011b] Stránský Jan, Milan Jirásek (2011), **Inelastic Calibration of Particle Models using Cells with Periodic Boundary Conditions**. In *Nano and Macro Mechanics 2011*.

[Stransky2012a] Stránský Jan, Milan Jirásek (2012), **Review on evaluation fo equivalent stress tensor in the discrete element method**. In *Nano and Macro Mechanics 2012*.

[Stransky2013a] Stránský Jan, Milan Jirásek (2013), **Localization Analysis of a Discrete Element Periodic Cell**. In *Nano and Macro Mechanics 2013*.

[Stransky2012b] Stránský Jan, Milan Jirásek (2012), **Open Source FEM-DEM Coupling**. In *Engineering Mechanics 2012*.

[Stransky2014a] Stránský Jan (2014), **Stochastic Wang tiles generation using DEM and YADE software**. In *Nano and Macro Mechanics 2014*.

[Stransky2013b] Stránský Jan (2013), **FEM - DEM Coupling and MuPIF Framework**. In *Engineering Mechanics 2013*.

[Stransky2014b] Stránský Jan (2014), **Combination of FEM and DEM with Application to Railway Ballast-Sleeper Interaction**. In *Engineering Mechanics 2014*.

[Stransky2014c] Stránský Jan (2014), **Kombinace MDP a MKP pro modelování interakce mezi železničním pražcem a podložím**. In *Juniorstav 2014*.

[Stransky2013c] Stránský Jan (2013), **Open Source DEM-FEM Coupling**. In *Particles 2013*.

[Suhr2015] Suhr, Bettina, Six, Klaus (2015), **Tribological effects in granular materials and their impact on the macroscopic material behaviour**. In *Proceedings of the IV International Conference on Particle-based Methods (PARTICLES 2015)*.

[Sweijen2014] Sweijen Thomas, Majid Hassanizadeh, Bruno Chareyre (2014), **Pore-Scale modeling of swelling porous media; application to super absorbent polymers**. In *XX . International Conference on Computational Methods in Water Resources*.

[Sweijen2017c] Sweijen, T, Hassanizadeh, SM, Aslannejad, H, Leszczynski, S (2017), **Grain-Scale Modelling of Swelling Granular Materials Using the Discrete Element Method and the Multi-Sphere Approximation**. In *Sixth Biot Conference on Poromechanics*. DOI 10.1061/9780784480779.040

[Tejada2016b] Tejada, IG, Sibille, L, Chareyre, B, Vincens, E (2016), **Numerical modeling of particle migration in granular soils**. In *Scour and Erosion: Proceedings of the 8th International Conference on Scour and Erosion (Oxford, UK, 12-15 September 2016)*.

[Tejada2017] Tejada, Ignacio G, Sibille, Luc, Chareyre, Bruno, Zhong, Chuheng, Marot, Didier (2017), **Multiscale modeling of transport of grains through granular assemblies**. In *EPJ Web of Conferences*.

[Tejchman2011] Tejchman, J. (2011), **Comparative Modelling of Shear Zone Patterns in Granular Bodies with Finite and Discrete Element Model**. *Advances in Bifurcation and Degradation in Geomaterials*, pages 255–260.

[Thoeni2015] Thoeni K., A. Giacomini, C. Lambert, S.W. Sloan (2015), **Rockfall Trajectory Analysis with Drapery Systems**. In *Engineering Geology for Society and Territory - Volume 2* (G. Lollino and D. Giordan and G.B. Crosta and J. Corominas and R. Azzam and J. Wasowski and N. Sciarra, ed.), Springer , DOI 10.1007/978-3-319-09057-3_356

[Thoeni2011] Thoeni K., C. Lambert, A. Giacomini, S.W. Sloan (2011), **Discrete Modelling of a Rockfall Protective System**. In *Particles 2011: Fundamentals and Applications*.

[Toraldo2015] Toraldo, Marcella, Chareyre, Bruno, Sibille, Luc (2015), **Numerical modelling of the localized fluidization in a saturated granular medium using the coupled method DEM-PFV**. In *Annual Report 1* (Grenoble Geomechanics Group, ed.),

[Tran2012b] Tran, V.D.H, Meguid, M.A, Chouinard, L.E (2012), **Coupling of Random Field Theory and the Discrete Element Method in the Reliability Analysis of Geotechnical Problems**. In *Canadian Society for Civil Engineering (CSCE) Annual Conference 2012, Edmonton*

[Tran2012d] Tran, V.D.H, Meguid, M.A, Chouinard, L.E (2012), **A Discrete Element Study of the Earth Pressure Distribution on Cylindrical Shafts**. In *Tunnelling Association of Canada (TAC) Conference 2012, Montreal*

[Uhlmann2014] Uhlmann, Eckart, Dethlefs, Arne, Eulitz, Alexander (2014), **Investigation of material removal and surface topography formation in vibratory finishing**. In *Procedia CIRP*.

[Wagner2023] W'agner, 'Arp'ad, Szab'o, Bence, Kov'acs, L'aszl'o, Tam'as, Korn'el, Grad-Gyenge, L'aszl'o (2023), **The development of a soil-movement measurement system to create more precise numeric soil models**. In *1st Workshop on Intelligent Infocommunication Networks, Systems and Services (WI2NS2)*.

[Wan2015b] Wan, Richard, Hadda, Nejib (2015), **Plastic Deformations in Granular Materials with Rotation of Principal Stress Axes**. In *Bifurcation and Degradation of Geomaterials in the New Millennium*.

[Yuan2014] Yuan Chao, Bruno Chareyre, Félix Darve (2014), **Pore-Scale Simulations of Drainage for Two-phase Flow in Dense Sphere Packings**. In *XX . International Conference on Computational Methods in Water Resources*.

[Yuan2017b] Yuan, Chao, Chareyre, Bruno, Darve, Felix (2017), **Coupled flow and deformations in granular systems beyond the pendular regime**. In *EPJ Web of Conferences*.

[Zhang2025] Zhang, A, Collin, F (2025), **Effect of Cementation on the Cyclic Behaviour of Sands: A 3D DEM Investigation**. In *IOP Conference Series: Earth and Environmental Science*.

[Zimbrod2022] Zimbrod, Patrick, Schreter, Magdalena, Schilp, Johannes (2022), **Efficient Simulation of Complex Capillary Effects in Advanced Manufacturing Processes using the Finite Volume Method**. In *2022 International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*.

[Abdallah2023] Abdallah, Ali (2023), **Filtration in granular materials with non-spherical particle shapes**. PhD thesis at *Ecole Centrale de Lyon*.

[Angelidakis2022] Angelidakis, Vasileios (2022), **Image-informed numerical modelling of particulate systems with irregular grains.**. PhD thesis at *Newcastle University*.

[Audry2023] Audry, Nils (2023), **Mod'elisation microm'ecanique de la densification d'un milieu granulaire coh'esif constitu'e de particules ductiles**. PhD thesis at *Universit'e Grenoble Alpes*.

[Benniou2016] Benniou Hicham (2016), **Modélisation par éléments discrets du comportement des matériaux cimentaires sous impact sévère : prise en compte du taux de saturation**. PhD thesis at *École doctorale Ingénierie - matériaux mécanique énergétique environnement procédés production (Grenoble)*.

[Boon2013b] Boon C.W. (2013), **Distinct Element Modelling of Jointed Rock Masses: Algorithms and Their Verification**. PhD thesis at *University of Oxford*.

[Borrmann2014] Borrmann Sebastian (2014), **DEM-CFD Simulation: Erprobung neuer Kopplungsansätze in ausgewählten Softwarepaketen (in german)**. Master thesis at *Institute of Mechanics and Fluid Dynamics, TU Bergakademie Freiberg*.

[Boschi2022] Boschi, Katia (2022), **Permeation grouting in granular materials. From micro to macro, from experimental to numerical and viceversa**. PhD thesis at *Politecnico di Milano*.

[Catalano2008a] Catalano E. (2008), **Infiltration effects on a partially saturated slope - An application of the Discrete Element Method and its implementation in the open-source software YADE**. Master thesis at *UJF-Grenoble*.

[Catalano2012] Catalano Emanuele (2012), **A pore-scale coupled hydromechanical model for biphasic granular media**. PhD thesis at *Université de Grenoble*.

[Caulk2022] Caulk, Robert Alexander (2022), **Mod'elisation micro-macro par la m'ethode des 'el'ements discrets (DEM) du comportement 'a long terme des scellements de puits sous sollicitation hydraulique-gaz**. PhD thesis at *Universit'e Grenoble Alpes*.

[Dedieu2025] Dedieu, Benjamin (2025), **Experimental and numerical modelling of vertical segregation in bedload sediment transport: size ratio effect on the ascent of an intruder**. PhD thesis at *Universit'e Grenoble Alpes [2020-….]*.

[Guo2025] Guo, Chang, others (2025), **Axial soil-pipeline interaction under different soil moisture conditions: physical and dem modelling**. PhD thesis at *Hong Kong Polytechnic University*.

[Charlas2013] Charlas Benoit (2013), **Etude du comportement mécanique d'un hydrure intermétallique utilisé pour le stockage d'hydrogène**. PhD thesis at *Université de Grenoble*.

[Chen2009a] Chen, F. (2009), **Coupled Flow Discrete Element Method Application in Granular Porous Media using Open Source Codes**. PhD thesis at *University of Tennessee, Knoxville*.

[Chen2011b] Chen, J. (2011), **Discrete Element Method (DEM) Analyses for Hot-Mix Asphalt (HMA) Mixture Compaction**. PhD thesis at *University of Tennessee, Knoxville*.

[Cusmano2023] Cusmano, Valeria (2023), **Out-of-plane seismic response of Unreinforced Masonry structures: a Discrete Macro-Element Approach including P-Delta effects**. PhD thesis at *Universit'a degli studi di Catania*.

[delValle2023] Carlos Andr'es del Valle (2023), **Development of a DEM-Based Model for the Simulation of Fractures in Brittle Materials**. Master thesis at *National University of Colombia*. DOI https://doi.org/10.5281/zenodo.10070737

[Deng2022] Deng, Na (2022), **Microm'ecanique de l''etat critique et son 'emergence dans la mod'elisation multi-'echelle de la rupture des sols**. PhD thesis at *Universit'e Grenoble Alpes*.

[DePue2019c] De Pue, Jan (2019), **Advances in modelling vehicle-induced stress transmission in relation to soil compaction**. PhD thesis at *Ghent University*.

[Duriez2009a] Duriez J. (2009), **Stabilité des massifs rocheux : une approche mécanique**. PhD thesis at *Institut polytechnique de Grenoble*.

[Duverger2023] Duverger, Sacha (2023), **A multi-scale, MPMxDEM, numerical modelling approach for geotechnical structures under severe loading**. PhD thesis at *Aix-Marseille Université (AMU)*.

[Favier2009c] Favier, L. (2009), **Approche numérique par éléments discrets 3D de la sollicitation d'un écoulement granulaire sur un obstacle**. PhD thesis at *Université Grenoble I – Joseph Fourier*.

[GaeblerMedack2013] Gäbler, Nils, Medack, Jörg (2013), **Experimental and simulative study of particle dynamics on a chute**. Project work at *Institute of Mechanics and Fluid Dynamics, TU Bergakademie Freiberg*.

[GentzAverhausVilbusch2014] Gentz, Julia, Averhaus, Jan, Vilbusch, Stephan (2014), **Numerical simulation of particle movement on a pelletizing disc – set-up and validation of a DEM model**. Project work at *Institute of Mechanics and Fluid Dynamics, TU Bergakademie Freiberg*.

[Gladky2019] Gladky, Anton (2019), **Numerische Untersuchung der Beanspruchung in Gutbettwalzenmühlen mit idealisierten Materialien (in german)**. PhD thesis at *Institute of Mineral Processing Machines, TU Bergakademie Freiberg*.

[Grabowski2024] Grabowski, Aleksander (2024), **Numerical investigation of mesostructural phenomena in shear zone during confined granular flow in a laboratory-scale silo using Discrete Element Method**. PhD thesis at *Gda'nsk University of Technology*.

[Guo2014c] Guo N. (2014), **Multiscale characterization of the shear behavior of granular media**. PhD thesis at *The Hong Kong University of Science and Technology*.

[Haustein2021a] Haustein, Martin (2021), **Beitrag zur Untersuchung von Partikelinteraktionen in Suspensionen am Beispiel von Stahl und Beton (in german)**. PhD thesis at *Institute of Mechanics and Fluid Dynamics, TU Bergakademie Freiberg*.

[Jakob2016] Jakob Christian (2016), **Numerische Modellierung des Verflüssigungsverhaltens von Kippen des Braunkohlenbergbaus beim und nach dem Wiederaufgang von Grundwasser (in german with extended summary in english)**. PhD thesis at *TU Bergakademie Freiberg*.

[Jerier2009b] Jerier, J.F. (2009), **Modélisation de la compression haute densité des poudres métalliques ductiles par la méthode des éléments discrets (in french)**. PhD thesis at *Université Grenoble I – Joseph Fourier*.

[Khosravani2014] Khosravani S. (2014), **An Effective Stress Equation for Unsaturated Granular Media in Pendular Regime**. Master thesis at *Department of Civil Engineering, University of Calgary*.

[Klichowicz2021a] Klichowicz, Michael (2021), **Modeling of realistic microstructures on the basis of quantitative mineralogical analyses**. PhD thesis at *Institute for Mineral Processing Machines and Recycling Systems Technology, TU Bergakademie Freiberg*.

[Kozicki2007b] Kozicki J. (2007), **Application of Discrete Models to Describe the Fracture Process in Brittle Materials**. PhD thesis at *Gdansk University of Technology*.

[Kunhappan2018] Kunhappan, Deepak (2018), **Numerical modeling of long flexible fibers in inertial flows.**. PhD thesis at *Université Grenoble Alpes*.

[Marzougui2011] Marzougui, D. (2011), **Hydromechanical modeling of the transport and deformation in bed load sediment with discrete elements and finite volume**. Master thesis at *Ecole Nationale d'Ingénieur de Tunis*.

[Maurin2015PhD] Maurin Raphael (2015), **Investigation of granular behavior in bedload transport using an Eulerian-Lagrangian model**. PhD thesis at *Université Grenoble Alpes*.

[Medack2014] Medack, Jörg (2014), **Untersuchungen zur Beeinflussung der örtlichen Aufgabegutverteilung auf der Schurre eines Hammerbrechers (in german)**. Master thesis at *Institute for Mineral Processing Machines, TU Bergakademie Freiberg*.

[Morales2012a] Morales D. (2012), **Cave Back Estimation Through Discrete Element Method, Based on Production Information**. Master thesis at *Universidad de Chile*.

[Pircher2021] Pircher Paul (2021), **A DEM model for elastic sleepers to study dynamic railway track behaviour**. Master thesis at *Montanuniversität Leoben*. DOI 10.34901/mul.pub.2021.5

[Scholtes2009d] Scholtès Luc (2009), **Modélisation micromécanique des milieux granulaires partiellement saturés**. PhD thesis at *Institut National Polytechnique de Grenoble*.

[Smilauer2010] vSmilauer, V'aclav (2010), **Cohesive particle model using discrete element method on the Yade platform**. PhD thesis at *Czech Technical University in Prague, Faculty of Civil Engineering & Université Grenoble I – Joseph Fourier, École doctorale I-MEP2*. (LaTeX sources)

[Smilauer2010c] Šmilauer Václav (2010), **Doctoral thesis statement**. *(PhD thesis summary)*. (LaTeX sources)

[Stein2017] Stein Gerhard Wolfgang (2017), **Design und Implementierung einer modularisierten Diskrete Elemente Methode Software**. PhD thesis at *Ruhr-Universität Bochum*.

[Stransky2018] Stránský Jan (2018), **Mesoscale Discrete Element Model for Concrete and Its Combination with FEM**. PhD thesis at *Czech Technical University in Prague*.

[Tran2011b] TRAN Van Tieng (2011), **Structures en béton soumises à des chargements mécaniques extrêmes: modélisation de la réponse locale par la méthode des éléments discrets (in french)**. PhD thesis at *Université Grenoble I – Joseph Fourier*.

[Torres2024] Torres Rodr'iguez, Ginna Marcela (2024), **Efecto del uso de barreras flexibles en el comportamiento de un flujo de detritos aplicado a la cuenca alta de la quebrada Taruca, municipio de Mocoa, utilizando DEM**. PhD thesis at *Universidad Nacional de Colombia*.

[VanDerHaven2024] van der Haven, Dingeman Lambertus Hendrik (2024), **On the Compaction of Granular Matter; Continuum and Discrete Numerical Modelling**. PhD thesis at *University of Cambridge (United Kingdom)*.

[Maurin2018a] Maurin, R. (2018), **YADE 1D vertical VANS fluid resolution: Numerical resolution details**. *Yade Technical Archive*.

[Maurin2018b] Maurin, R. (2018), **YADE 1D vertical VANS fluid resolution: Theoretical basis**. *Yade Technical Archive*.

[Maurin2018c] Maurin, R. (2018), **YADE 1D vertical VANS fluid resolution: validations**. *Yade Technical Archive*.

[Chareyre2019b] Chareyre, B., Caulk, R.A., Chèvremont, W., Guntz, T., Kneib, F., Kunhappen, D., Pourroy, J. (2019), **Calcul distribué MPI pour la dynamique de systèmes particulaires**. *Yade Technical Archive*.

[Pirnia2019] Pirnia, P. (2019), **COMSOL-Yade Interface (ICY) instruction guide**. *Yade Technical Archive*.

[Caulk2018] Caulk, R. (2018), **Stochastic Augmentation of the Discrete Element Method for the Investigation of Tensile Rupture in Heterogeneous Rock**. *Yade Technical Archive*. DOI 10.5281/zenodo.1202039

[Bertrand2008] D. Bertrand, F. Nicot, P. Gotteland, S. Lambert (2008), **Discrete element method (DEM) numerical modeling of double-twisted hexagonal mesh **. *Canadian Geotechnical Journal* (45), pages 1104–1117.

[Bertrand2005] D. Bertrand, F. Nicot, P. Gotteland, S. Lambert (2005), **Modelling a geo-composite cell using discrete analysis**. *Computers and Geotechnics* (32), pages 564–577.

[Chan2011] D. Chan, E. Klaseboer, R. Manica (2011), **Film drainage and coalescence between deformable drops and bubbles.**. *Soft Matter* (7), pages 2235-2264.

[Chareyre2002a] B. Chareyre, L. Briancon, P. Villard (2002), **Theoretical versus experimental modeling of the anchorage capacity of geotextiles in trenches.**. *Geosynthet. Int.* (9), pages 97–123.

[Chareyre2005] Bruno Chareyre, Pascal Villard (2005), **Dynamic Spar Elements and Discrete Element Methods in Two Dimensions for the Modeling of Soil-Inclusion Problems**. *Journal of Engineering Mechanics* (131), pages 689–698. DOI 10.1061/(ASCE)0733-9399(2005)131:7(689)

[Chareyre2003] Bruno Chareyre (2003), **Modélisation du comportement d'ouvrages composites sol-géosynthétique par éléments discrets - Application aux tranchées d'ancrage en tête de talus.**. PhD thesis at *Grenoble University*.

[Chareyre2002b] B. Chareyre, P. Villard (2002), **Discrete element modeling of curved geosynthetic anchorages with known macro-properties.**. In *Proc., First Int. PFC Symposium, Gelsenkirchen, Germany*.

[Chevremont2019] Chèvremont, William, Chareyre, Bruno, Bodiguel, Hugues (2019), **Quantitative study of the rheology of frictional suspensions: Influence of friction coefficient in a large range of viscous numbers**. *Phys. Rev. Fluids* 6 (4), pages 064302. DOI 10.1103/PhysRevFluids.4.064302

[Chevremont2020] William Chèvremont, Hugues Bodiguel, Bruno Chareyre (2020), **Lubricated contact model for numerical simulations of suspensions**. *Powder Technology* (372), pages 600 - 610. DOI https://doi.org/10.1016/j.powtec.2020.06.001

[Villard2004a] P. Villard, B. Chareyre (2004), **Design methods for geosynthetic anchor trenches on the basis of true scale experiments and discrete element modelling**. *Canadian Geotechnical Journal* (41), pages 1193–1205.

[Lu1998] Ya Yan Lu (1998), **Computing the Logarithm of a Symmetric Positive Definite Matrix**. *Appl. Numer. Math* (26), pages 483–496. DOI 10.1016/S0168-9274(97)00103-7

[Alonso2004] F. Alonso-Marroquin, R. Garcia-Rojo, H.J. Herrmann (2004), **Micro-mechanical investigation of the granular ratcheting**. In *Cyclic Behaviour of Soils and Liquefaction Phenomena*.

[McNamara2008] S. McNamara, R. García-Rojo, H. J. Herrmann (2008), **Microscopic origin of granular ratcheting**. *Physical Review E* (77). DOI 10.1103/PhysRevE.77.031304

[GarciaRojo2004] R. García-Rojo, S. McNamara, H. J. Herrmann (2004), **Discrete element methods for the micro-mechanical investigation of granular ratcheting**. In *Proceedings ECCOMAS 2004*.

[Allen1989] M. P. Allen, D. J. Tildesley (1989), **Computer simulation of liquids**. Clarendon Press.

[DeghmReport2006] F. V. Donzé (ed.), **Annual Report 2006** (2006). *Discrete Element Group for Hazard Mitigation*. Université Joseph Fourier, Grenoble

[Pournin2001] L. Pournin, Th. M. Liebling, A. Mocellin (2001), **Molecular-dynamics force models for better control of energy dissipation in numerical simulations of dense granular media**. *Phys. Rev. E* (65), pages 011302. DOI 10.1103/PhysRevE.65.011302

[Jung1997] Derek Jung, Kamal K. Gupta (1997), **Octree-based hierarchical distance maps for collision detection**. *Journal of Robotic Systems* (14), pages 789–806. DOI 10.1002/(SICI)1097-4563(199711)14:11<789::AID-ROB3>3.0.CO;2-Q

[Hubbard1996] Philip M. Hubbard (1996), **Approximating polyhedra with spheres for time-critical collision detection**. *ACM Trans. Graph.* (15), pages 179–210. DOI 10.1145/231731.231732

[Klosowski1998] James T. Klosowski, Martin Held, Joseph S. B. Mitchell, Henry Sowizral, Karel Zikan (1998), **Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs**. *IEEE Transactions on Visualization and Computer Graphics* (4), pages 21–36.

[Munjiza2006] A. Munjiza, E. Rougier, N. W. M. John (2006), **MR linear contact detection algorithm**. *International Journal for Numerical Methods in Engineering* (66), pages 46–71. DOI 10.1002/nme.1538

[Munjiza1998] A. Munjiza, K. R. F. Andrews (1998), **NBS contact detection algorithm for bodies of similar size**. *International Journal for Numerical Methods in Engineering* (43), pages 131–149. DOI 10.1002/(SICI)1097-0207(19980915)43:1<131::AID-NME447>3.0.CO;2-S

[Verlet1967] Loup Verlet (1967), **Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules**. *Phys. Rev.* (159), pages 98. DOI 10.1103/PhysRev.159.98

[Luding2008] Stefan Luding (2008), **Introduction to discrete element methods**. In *European Journal of Environmental and Civil Engineering*.

[Wang2009] Yucang Wang (2009), **A new algorithm to model the dynamics of 3-D bonded rigid bodies with rotations**. *Acta Geotechnica* (4), pages 117–127. DOI 10.1007/s11440-008-0072-1

[Omelyan1999] Igor P. Omelyan (1999), **A New Leapfrog Integrator of Rotational Motion. The Revised Angular-Momentum Approach**. *Molecular Simulation* (22). DOI 10.1080/08927029908022097

[Omelyan1998] Igor P. Omelyan (1998), **Algorithm for numerical integration of the rigid-body equations of motion**. *Physical Review E* (58), pages 1169–1172. DOI 10.1103/physreve.58.1169

[Fincham1992] David Fincham (1992), **Leapfrog Rotational Algorithms**. *Molecular Simulation* (8), pages 165-178. DOI 10.1080/08927029208022474

[Neto2006] Natale Neto, Luca Bellucci (2006), **A new algorithm for rigid body molecular dynamics**. *Chemical Physics* (328), pages 259–268. DOI 10.1016/j.chemphys.2006.07.009

[Johnson2008] Scott M. Johnson, John R. Williams, Benjamin K. Cook (2008), **Quaternion-based rigid body rotation integration algorithms for use in particle methods**. *International Journal for Numerical Methods in Engineering* (74), pages 1303–1313. DOI 10.1002/nme.2210

[Addetta2001] G.A. D'Addetta, F. Kun, E. Ramm, H.J. Herrmann (2001), **From solids to granulates - Discrete element simulations of fracture and fragmentation processes in geomaterials.**. In *Continuous and Discontinuous Modelling of Cohesive-Frictional Materials*.

[Pfc3dManual30] ICG (2003), **PFC3D (Particle Flow Code in 3D) Theory and Background Manual, version 3.0**. Itasca Consulting Group.

[Hentz2003] Séebastien Hentz (2003), **Modélisation d'une Structure en Béton Armé Soumise à un Choc par la méthode des Eléments Discrets**. PhD thesis at *Université Grenoble 1 – Joseph Fourier*.

[Price2007] Mathew Price, Vasile Murariu, Garry Morrison (2007), **Sphere clump generation and trajectory comparison for real particles**. In *Proceedings of Discrete Element Modelling 2007*.

[Kuhl2001] E. Kuhl, G. A. D'Addetta, M. Leukart, E. Ramm (2001), **Microplane modelling and particle modelling of cohesive-frictional materials**. In *Continuous and Discontinuous Modelling of Cohesive-Frictional Materials*. DOI 10.1007/3-540-44424-6_3

[Thornton1991] Colin Thornton, K. K. Yin (1991), **Impact of elastic spheres with and without adhesion**. *Powder technology* (65), pages 153–166. DOI 10.1016/0032-5910(91)80178-L

[Thornton2000] Colin Thornton (2000), **Numerical simulations of deviatoric shear deformation of granular media**. *Géotechnique* (50), pages 43–53. DOI 10.1680/geot.2000.50.1.43

[Thornton2013] Colin Thornton, Sharen J. Cummins, Paul W. Cleary (2013), **An investigation of the comparative behaviour of alternative contact force models during inelastic collisions**. *Powder Technology* (233), pages 30-46. DOI 10.1016/j.powtec.2012.08.012

[Thornton2015] Thornton, Colin (2015), **Granular dynamics, contact mechanics and particle system simulations**. *A DEM study. Particle Technology Series* (24). DOI 10.1007/978-3-319-18711-2

[CundallStrack1979] P.A. Cundall, O.D.L. Strack (1979), **A discrete numerical model for granular assemblies**. *Geotechnique* (), pages 47–65. DOI 10.1680/geot.1979.29.1.47

[cgal] Jean-Daniel Boissonnat, Olivier Devillers, Sylvain Pion, Monique Teillaud, Mariette Yvinec (2002), **Triangulations in CGAL**. *Computational Geometry: Theory and Applications* (22), pages 5–19.

[Satake1982] M. Satake (1982), **Fabric tensor in granular materials.**. In *Proc., IUTAM Symp. on Deformation and Failure of Granular materials, Delft, The Netherlands*.

[Hentz2004a] S. Hentz, F.V. Donzé, L.Daudeville (2004), **Discrete element modelling of concrete submitted to dynamic loading at high strain rates**. *Computers and Structures* (82), pages 2509–2524. DOI **'10.1016/j.compstruc.2004.05.016 <http://dx.doi.org/10.1016/j.compstruc.2004.05.016 >'_**

[Hentz2004b] S. Hentz, L. Daudeville, F.V. Donzé (2004), **Identification and Validation of a Discrete Element Model for Concrete**. *ASCE Journal of Engineering Mechanics* (130), pages 709–719. DOI 10.1061/(ASCE)0733-9399(2004)130:6(709)

[Camborde2000a] F. Camborde, C. Mariotti, F.V. Donzé (2000), **Numerical study of rock and Concrete behaviour by discrete element modelling**. *Computers and Geotechnics* (27), pages 225–247.

[Donze1999a] F.V. Donzé, S.A. Magnier, L. Daudeville, C. Mariotti, L. Davenne (1999), **Study of the behavior of concrete at high strain rate compressions by a discrete element method**. *ASCE J. of Eng. Mech* (125), pages 1154–1163. DOI 10.1016/S0266-352X(00)00013-6

[Magnier1998a] S.A. Magnier, F.V. Donzé (1998), **Numerical simulation of impacts using a discrete element method**. *Mech. Cohes.-frict. Mater.* (3), pages 257–276. DOI 10.1002/(SICI)1099-1484(199807)3:3<257::AID-CFM50>3.0.CO;2-Z

[Donze1995a] F.V. Donzé, S.A. Magnier (1995), **Formulation of a three-dimensional numerical model of brittle behavior**. *Geophys. J. Int.* (122), pages 790–802.

[Donze1994a] F.V. Donzé, P. Mora, S.A. Magnier (1994), **Numerical simulation of faults and shear zones**. *Geophys. J. Int.* (116), pages 46–52.

[Hentz2005a] S. Hentz, F.V. Donzé, L.Daudeville (2005), **Discrete elements modeling of a reinforced concrete structure submitted to a rock impact**. *Italian Geotechnical Journal* (XXXIX), pages 83–94.

[Donze2004a] F.V. Donzé, P. Bernasconi (2004), **Simulation of the Blasting Patterns in Shaft Sinking Using a Discrete Element Method**. *Electronic Journal of Geotechnical Engineering* (9), pages 1–44.

[Kettner2011] Lutz Kettner, Andreas Meyer, Afra Zomorodian (2011), **Intersecting Sequences of dD Iso-oriented Boxes**. In *CGAL User and Reference Manual*.

[Pion2011] Sylvain Pion, Monique Teillaud (2011), **3D Triangulations**. In *CGAL User and Reference Manual*.

[Calvetti1997] Calvetti, F., Combe, G., Lanier, J. (1997), **Experimental micromechanical analysis of a 2D granular material: relation between structure evolution and loading path**. *Mechanics of Cohesive-frictional Materials* (2), pages 121–163.

[Bagi2006] Katalin Bagi (2006), **Analysis of microstructural strain tensors for granular assemblies**. *International Journal of Solids and Structures* (43), pages 3166 - 3184. DOI 10.1016/j.ijsolstr.2005.07.016

[Brilliantov1996] Brilliantov, Nikolai V, Spahn, Frank, Hertzsch, Jan-Martin, P"oschel, Thorsten (1996), **Model for collisions in granular gases**. *Physical review E* (53), pages 5382.

[Weigert1999] Weigert, Tom, Ripperger, Siegfried (1999), **Calculation of the Liquid Bridge Volume and Bulk Saturation from the Half-filling Angle**. *Particle & Particle Systems Characterization* (16), pages 238–242. DOI 10.1002/(SICI)1521-4117(199910)16:5<238::AID-PPSC238>3.0.CO;2-E

[Willett2000] Willett, Christopher D., Adams, Michael J., Johnson, Simon A., Seville, Jonathan P. K. (2000), **Capillary Bridges between Two Spherical Bodies**. *Langmuir* (16), pages 9396-9405. DOI 10.1021/la000657y

[Herminghaus2005] Herminghaus, S. (2005), **Dynamics of wet granular matter**. *Advances in Physics* (54), pages 221-261. DOI 10.1080/00018730500167855

[Rabinov2005] RABINOVICH Yakov I., ESAYANUR Madhavan S., MOUDGIL Brij M. (2005), **Capillary forces between two spheres with a fixed volume liquid bridge : Theory and experiment**. *Langmuir* (21), pages 10992–10997. (eng)

[Luding2008b] Luding, Stefan (2008), **Cohesive, frictional powders: contact models for tension**. *Granular Matter* (10), pages 235-246. DOI 10.1007/s10035-008-0099-x

[Zhou1999536] Y.C. Zhou, B.D. Wright, R.Y. Yang, B.H. Xu, A.B. Yu (1999), **Rolling friction in the dynamic simulation of sandpile formation**. *Physica A: Statistical Mechanics and its Applications* (269), pages 536–553. DOI 10.1016/S0378-4371(99)00183-1

[Antypov2011] D. Antypov, J. A. Elliott (2011), **On an analytical solution for the damped Hertzian spring**. *EPL (Europhysics Letters)* (94), pages 50004.

[Ivars2011] Diego Mas Ivars, Matthew E. Pierce, Caroline Darcel, Juan Reyes-Montes, David O. Potyondy, R. Paul Young, Peter A. Cundall (2011), **The synthetic rock mass approach for jointed rock mass modelling **. *International Journal of Rock Mechanics and Mining Sciences * (48), pages 219 - 244. DOI 10.1016/j.ijrmms.2010.11.014

[Potyondy2004] D.O. Potyondy, P.A. Cundall (2004), **A bonded-particle model for rock **. *International Journal of Rock Mechanics and Mining Sciences * (41), pages 1329 - 1364. DOI 10.1016/j.ijrmms.2004.09.011

[Radjai2011] Radjai, F., Dubois, F. (2011), **Discrete-Element Modeling of Granular Materials**. John Wiley & Sons.

[Schwager2007] Schwager, Thomas, Pöschel, Thorsten (2007), **Coefficient of restitution and linear–dashpot model revisited**. *Granular Matter* (9), pages 465-469. DOI 10.1007/s10035-007-0065-z

[Lambert2008] Lambert, Pierre, Chau, Alexandre, Delchambre, Alain, Régnier, Stéphane (2008), **Comparison between two capillary forces models**. *Langmuir* (24), pages 3157–3163.

[Mueller2003] Müller, Matthias, Charypar, David, Gross, Markus (2003), **Particle-based Fluid Simulation for Interactive Applications**. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*.

[Mueller2011] M"uller, Patric, P"oschel, Thorsten (2011), **Collision of viscoelastic spheres: Compact expressions for the coefficient of normal restitution**. *Physical Review E* (84), pages 021302.

[Soulie2006] Soulié, F., Cherblanc, F., El Youssoufi, M.S., Saix, C. (2006), **Influence of liquid bridges on the mechanical behaviour of polydisperse granular materials**. *International Journal for Numerical and Analytical Methods in Geomechanics* (30), pages 213–228. DOI 10.1002/nag.476

[Mani2013] Mani, Roman, Kadau, Dirk, Herrmann, HansJ. (2013), **Liquid migration in sheared unsaturated granular media**. *Granular Matter* (15), pages 447-454. DOI 10.1007/s10035-012-0387-3

[Richardson1954] Richardson, J. F.,, W. N. Zaki (1954), **Sedimentation and fluidization: Part i**. *Trans. Instn. Chem. Engrs* (32).

[RevilBaudard2013] Revil-Baudard, T., Chauchat, J. (2013), **A two-phase model for sheet flow regime based on dense granular flow rheology**. *Journal of Geophysical Research: Oceans* (118), pages 619–634.

[RevilBaudard2015] Revil-Baudard, T., Chauchat, J., Hurther, D., Barraud, P-A. (2015), **Investigation of sheet-flow processes based on novel acoustic high-resolution velocity and concentration measurements**. *Journal of Fluid Mechanics* (767), pages 1–30. DOI 10.1017/jfm.2015.23

[Li1995] Li, L., Sawamoto, M. (1995), **Multi-phase model on sediment transport in sheet-flow regime under oscillatory flow**. *Coastal engineering Japan* (38), pages 157-178.

[Schmeeckle2007] Schmeeckle, Mark W., Nelson, Jonathan M., Shreve, Ronald L. (2007), **Forces on stationary particles in near-bed turbulent flows**. *Journal of Geophysical Research: Earth Surface* (112). DOI 10.1029/2006JF000536

[Wiberg1985] Wiberg, Patricia L., Smith, J. Dungan (1985), **A theoretical model for saltating grains in water**. *Journal of Geophysical Research: Oceans* (90), pages 7341–7354.

[Dallavalle1948] J. M. DallaValle (1948), **Micrometrics : The technology of fine particles**. Pitman Pub. Corp.

[Monaghan1992] Monaghan, J.~J. (1992), **Smoothed particle hydrodynamics**. *Annual Review of Astronomy and Astrophysics* (30), pages 543-574. DOI 10.1146/annurev.aa.30.090192.002551

[Morris1997] Morris, Joseph P, Fox, Patrick J, Zhu, Yi (1997), **Modeling Low Reynolds Number Incompressible Flows Using SPH**. *Journal of Computational Physics * (136), pages 214-226. DOI http://dx.doi.org/10.1006/jcph.1997.5776

[Lucy1977] Lucy, L.~B. (1977), **A numerical approach to the testing of the fission hypothesis**. *Astronomical Journal* (82), pages 1013-1024. DOI 10.1086/112164

[Monaghan1985] Monaghan, J.~J., Lattanzio, J.~C. (1985), **A refined particle method for astrophysical problems**. *Astronomy and Astrophysics* (149), pages 135-143.

[yade:doc] V. Šmilauer, E. Catalano, B. Chareyre, S. Dorofeenko, J. Duriez, A. Gladky, J. Kozicki, C. Modenese, L. Scholtès, L. Sibille, J. Stránský, K. Thoeni (2010), **Yade Documentation**. The Yade Project. (http://yade-dem.org/doc/)

[yade:manual] V. Šmilauer, A. Gladky, J. Kozicki, C. Modenese, J. Stránský (2010), **Yade, Using and Programming**. In *Yade Documentation* (V. Šmilauer, ed.), The Yade Project , 1st ed. (http://yade-dem.org/doc/)

[yade:background] V. Šmilauer, B. Chareyre (2010), **Yade DEM Formulation**. In *Yade Documentation* (V. Šmilauer, ed.), The Yade Project , 1st ed. (http://yade-dem.org/doc/formulation.html)

[yade:reference] V. Šmilauer, E. Catalano, B. Chareyre, S. Dorofeenko, J. Duriez, A. Gladky, J. Kozicki, C. Modenese, L. Scholtès, L. Sibille, J. Stránský, K. Thoeni (2010), **Yade Reference Documentation**. In *Yade Documentation* (V. Šmilauer, ed.), The Yade Project , 1st ed. (http://yade-dem.org/doc/)

[Tonon2005] Tonon, F. (2005), **Explicit Exact Formulas for the 3-D Tetrahedron Inertia Tensor in Terms of its Vertex Coordinates**. *Journal of mathematics and statistics* (1), pages 135-143.

[Lourenco1994] P B Lourenço 1994 **Analysis of masonry structures with interface elements. theory and applications** (Report No. 03-21-22-0-01, Delft University of Technology, Faculty of Civil Engineering)

[Hazzard2000] Hazzard, J.F, Young, R.P (2000), **Simulating acoustic emissions in bonded-particle models of rock**. *International Journal of Rock Mechanics and Mining Sciences* (37), pages 867-872. DOI 10.1016/S1365-1609(00)00017-4

[Hazzard2013] Hazzard, J. F., Damjanac, Branko (2013), **Further investigations of microseismicity in bonded particle models**. *3rd International FLAC/DEM Symposium*, pages 1-11.

[Scholz2003] Scholz, C. H., Harris, R. A. (2003), **The Mechanics of Earthquakes and Faulting**. Cambridge University Press.

[Jackson2000] R. Jackson (2000), **The dynamics of fluidized particles**. Cambridge University Press.

[Maurin2018_VANSbasis] R. Maurin (2018), **YADE 1D vertical VANS fluid resolution : Theoretical bases**. *Yade Technical Archive*.

[Maurin2018_VANSfluidResol] R. Maurin (2018), **YADE 1D vertical VANS fluid resolution : Fluid resolution details**. *Yade Technical Archive*.

[Maurin2018_VANSvalidations] R. Maurin (2018), **YADE 1D vertical VANS fluid resolution : validations**. *Yade Technical Archive*.

[Berger1987] Berger, M. J., Bokhari, S. H. (1987), **A Partitioning Strategy for Nonuniform Problems on Multiprocessors**. *IEEE Trans. Comput.* (36), pages 570-580. DOI 10.1109/TC.1987.1676942

[Fleissner2007] Fleissner, Florian, Eberhard, Peter (2007), **Parallel Load Balanced Particle Simulation with Hierarchical Particle Grouping Strategies**. In *IUTAM Symposium on Multiscale Problems in Multibody System Contacts*.

Python Module Index

_

yade._libVersions, 164
yade._log, 166
yade._math, 170
yade._minieigenHP, 211
yade._packObb, 308
yade._packPredicates, 308
yade._packSpheres, 303
yade._polyhedra_utils, 313
yade._utils, 331

b

yade.bf, 145
yade.bodiesHandling, 147

e

yade.export, 148

g

yade.geom, 153
yade.gridpfacet, 157

l

yade.libVersions, 161
yade.linterpolation, 165
yade.log, 166

m

yade.math, 168
yade.minieigenHP, 211
yade.mpy, 294

p

yade.pack, 298
yade.plot, 308
yade.polyhedra_utils, 312
yade.post2d, 314
yade.potential_utils, 317

t

yade.timing, 319

u

yade.utils, 321

y

yade.ymport, 342