

Yade: Using and Programming

1st edition

Editor

Václav Šmilauer

Authors

**Václav Šmilauer
Anton Gladky
Janek Kozicki
Chiara Modenese
Jan Stránský**

Authors

Václav Šmilauer

CVUT Prague - lab. 3SR Grenoble University

Anton Gladky

TU Bergakademie Freiberg

Janek Kozicki

Gdansk University of Technology - lab. 3SR Grenoble University

Chiara Modenese

University of Oxford

Jan Stránský

CVUT Prague

Citing this document

In order to let users cite Yade consistently in publications, we provide a list of bibliographic references for the different parts of the documentation. This way of acknowledging Yade is also a way to make developments and documentation of Yade more attractive for researchers, who are evaluated on the basis of citations of their work by others. We therefore kindly ask users to cite Yade as accurately as possible in their papers, as explained in <http://yade-dem/doc/citing.html>.

Abstract

Yade is an extensible open-source framework for discrete numerical models, focused on Discrete Element Method. The computation parts are written in c++ using flexible object model, allowing independent implementation of new algorithms and interfaces. Python is used for rapid and concise scene construction, simulation control, postprocessing and debugging. Yade interface and core design are presented. User's and programmer's manuals are included.

Keywords: Yade-DEM, design, c++, Python, open source, documentation.

Contents

1	Introduction	1
1.1	Getting started	1
1.2	Architecture overview	4
2	User's manual	11
2.1	Scene construction	11
2.2	Controlling simulation	27
2.3	Postprocessing	40
2.4	Python specialties and tricks	45
2.5	Extending Yade	45
2.6	Troubleshooting	45
3	Programmer's manual	49
3.1	Build system	49
3.2	Conventions	53
3.3	Support framework	57
3.4	Simulation framework	77
3.5	Runtime structure	82
3.6	Python framework	84
3.7	Maintaining compatibility	86
3.8	Debian packaging instructions	87
4	Installation	89
4.1	Packages	89
4.2	Source code	89
5	References	93
	Bibliography	95

Chapter 1

Introduction

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 topics, 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.

1.1.1 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 command²):

```
$ yade
Welcome to Yade b3r1984
TCP python prompt on localhost:9001, auth cookie `sdksuy'
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

- version (b3r1984); always state this version you use if you seek help in the community or report bug;
- some information for [Remote control](#), which you are unlikely to need now;
- basic help for the command-line that just appeared (Yade [1]:).

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.

Type `quit()`, `exit()` or simply press `^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 script.py
Welcome to Yade bzt1986
TCP python prompt on localhost:9001, auth cookie `askcsu'
TCP info provider on localhost:21000
Running script hello.py
Hello world!
[[ ^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.

1.1.2 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 [4]: TriaxialTest(numberOfGrains=200).load()
```

```
Yade [5]: len(O.bodies)
-> [5]: 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*.

1.1.3 Running simulation

As explained above, 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`:

```
Yade [7]: O.iter
-> [7]: 0
```

```
Yade [8]: O.time
-> [8]: 0.0
```

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.


```
Yade [9]: O.dt=1e-4
```

```
Yade [10]: O.step()
```

```
Yade [11]: O.iter  
-> [11]: 1
```

```
Yade [12]: O.time  
-> [12]: 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 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 [13]: O.run()
```

```
Yade [14]: O.pause()
```

```
Yade [15]: O.iter  
-> [15]: 1315
```

```
Yade [16]: O.run(100000,True)
```

```
Yade [17]: O.iter  
-> [17]: 101315
```

```
Yade [18]: O.stopAtIter=500000
```

```
Yade [19]: O.wait()
```

```
Yade [20]: O.iter  
-> [20]: 101315
```

1.1.4 Saving and loading

Simulation can be saved at any point to (optionally compressed) XML file. With some limitations, it is generally possible to load the XML later and resume the simulation as if it were not interrupted. Note that since XML is merely readable dump of Yade's internal objects, it might not (probably will not) open with different Yade version.

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

```
Yade [22]: O.reload()
```

```
Yade [24]: O.load('/tmp/another.xml.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 [25]: O.saveTmp()
```

```
Yade [26]: O.loadTmp()
```

```
Yade [27]: O.saveTmp('init') ## named memory slot
```

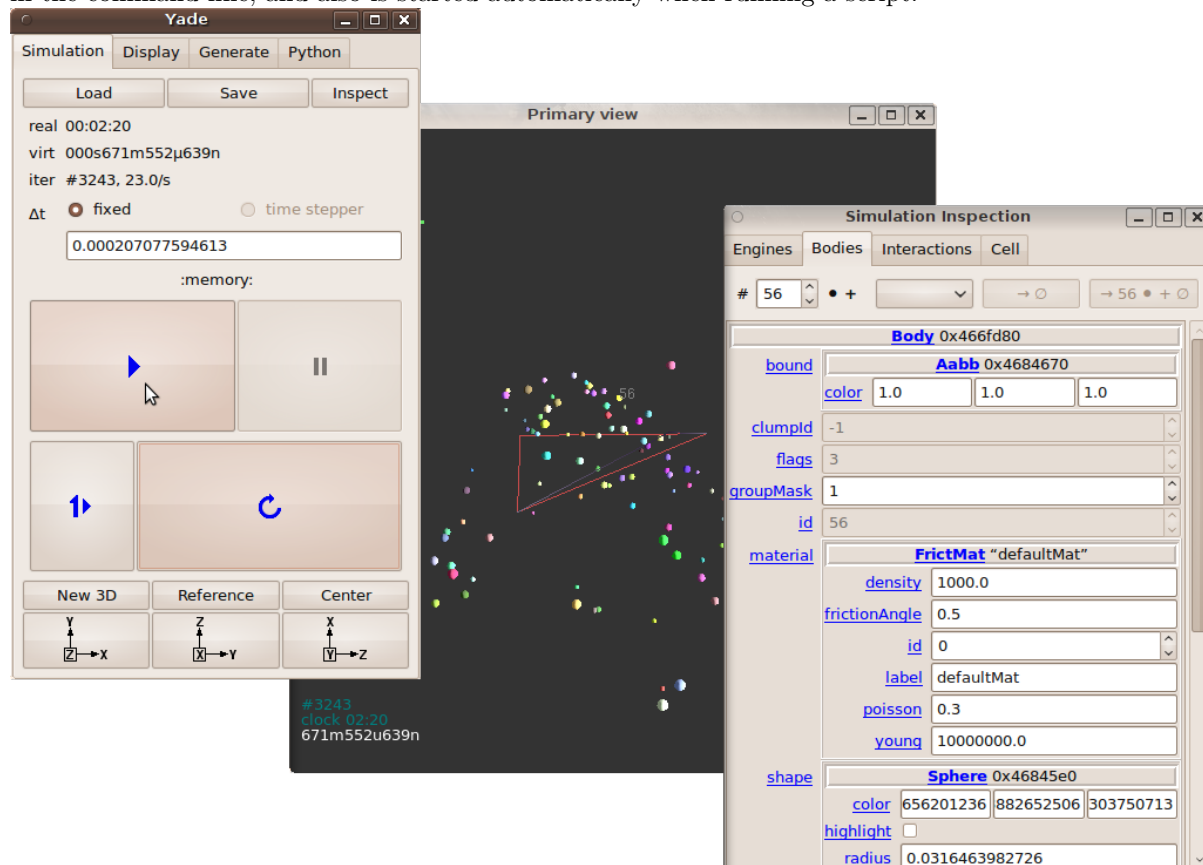
```
Yade [28]: O.loadTmp('init')
```

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.

1.1.5 Graphical interface

Yade can be optionally compiled with qt4-based graphical interface. It can be started by pressing **F12** in the command-line, and also is started automatically when running a script.



The windows with buttons is called **Controller** (can be invoked by `yade.qt.Controller()` from python):

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.

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.

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.

1.2.1 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 [32]: O.bodies          # some data components
-> [32]: <yade.wrapper.BodyContainer object at 0x67ab758>

Yade [33]: len(O.bodies)    # there are no bodies as of yet
-> [33]: 0

Yade [34]: O.engines        # functional components, empty at the moment
-> [34]: []
```

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 infinite **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 variable variables, in particular spatial **position** and **orientation**, **linear** and **angular** velocity, **linear** and **angular** accelerator; it is updated by the **integrator** at every step.

Derived classes can hold additional data, e.g. **averaged damage**.

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 **Dispatcher**’s 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 [35]: s=utils.sphere(center=[0,0,0],radius=1)

Yade [36]: s.shape, s.state, s.mat, s.bound
-> [36]:
```

```
(<Sphere instance at 0x8d9c160>,  
 <State instance at 0x61e5e00>,  
 <FrictMat instance at 0x512ca90>,  
 None)
```

```
Yade [37]: s.state.pos  
-> [37]: Vector3(0,0,0)
```

```
Yade [38]: s.shape.radius  
-> [38]: 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 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 [39]: O.bodies.append(s)  
-> [39]: 0
```

```
Yade [40]: O.bodies.append(utils.sphere([0,0,2],.5))  
-> [40]: 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 access this particular body later; it is remembered internally in `Body` itself. You can address bodies by their id:

```
Yade [41]: O.bodies[1].state.pos  
-> [41]: Vector3(0,0,2)
```

```
Yade [42]: O.bodies[100]
```

```
-----  
IndexError                                Traceback (most recent call last)  
  
/home/3S-LAB/bchareyre/yade/yade-test-kdev/yade/doc/sphinx/<ipython console> in <module>()  
  
IndexError: Body id out of range.
```

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

```
Yade [43]: O.bodies.append(s)
```

```
-----  
IndexError                                Traceback (most recent call last)  
  
/home/3S-LAB/bchareyre/yade/yade-test-kdev/yade/doc/sphinx/<ipython console> in <module>()  
  
IndexError: 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 [44]: 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**, **Facet+Sphere** etc.

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.

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

```
Yade [48]: O.interactions[0,1]
-> [48]: <Interaction instance at 0x57dde00>

Yade [49]: O.interactions[1,0]      # order of ids is not important
-> [49]: <Interaction instance at 0x57dde00>

Yade [50]: i=O.interactions[0,1]

Yade [51]: i.id1,i.id2
-> [51]: (0, 1)

Yade [52]: i.geom
-> [52]: <Dem3DofGeom_SphereSphere instance at 0x73fb600>

Yade [53]: i.phys
-> [53]: <FrictPhys instance at 0x98e4f80>
```

```
Yade [54]: O.interactions[100,10111]
```

```
-----
IndexError                                Traceback (most recent call last)
```

```
/home/3S-LAB/bchareyre/yade/yade-test-kdev/yade/doc/sphinx/<ipython console> in <module>()
```

```
IndexError: No such interaction
```

Generalized forces

Generalized forces include force, torque and forced displacement and rotation; they are stored only temporarily, 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 [55]: O.forces.f(0)
-> [55]: Vector3(0,0,0)

Yade [56]: O.forces.addF(0,Vector3(1,2,3))

Yade [57]: O.forces.f(0)
-> [57]: 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.

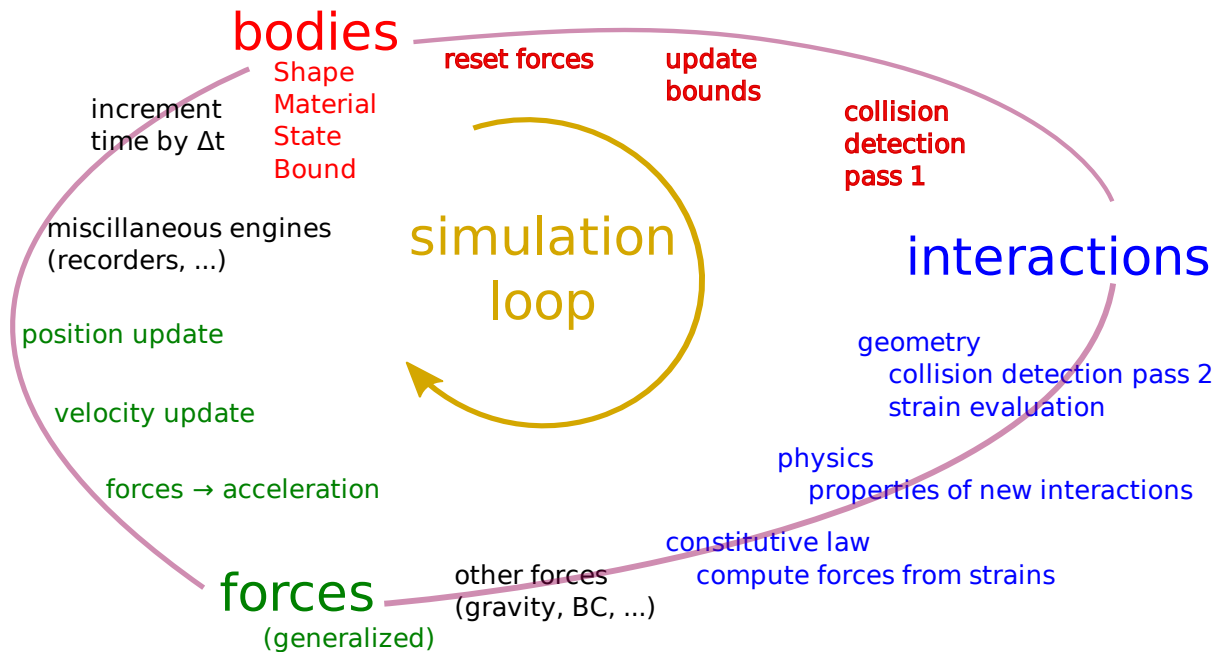


Figure 1.1: Typical simulation loop; each step begins at body-centered bit at 11 o'clock, continues with interaction bit, force application bit, miscellanea and ends with time update.

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

Engines

Simulation loop, shown at [img. img-yade-iter-loop](#), can be described as follows in Python (details will be explained later); each of the `O.engine` 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_Dem3DofGeom(),Ig2_Facet_Sphere_Dem3DofGeom()],
        [Ip2_FrictMat_FrictMat_FrictPhys()],
        [Law2_Dem3Dof_Elastic_Elastic()],
    ),
    # 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. [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 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 object 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. 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**.

The next part, reading

```
InteractionLoop(
    [Ig2_Sphere_Sphere_Dem3DofGeom(),Ig2_Facet_Sphere_Dem3DofGeom()],
    [Ip2_FrictMat_FrictMat_FrictPhys()],
    [Law2_Dem3Dof_Elastic_Elastic()],
),
```

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

IGeomDispatcher 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 **IGeom** (whence **Ig** in the name) associated with the interaction, if there is collision. The functor might as well fail on approximate interactions, indicating there is no real contact between the bodies, even if they did overlap in the approximate collision detection.

1. The first functor, **Ig2_Sphere_Sphere_Dem3DofGeom**, is called on interaction of 2 **Spheres** and creates **Dem3DofGeom** instance, if appropriate.
2. The second functor, **Ig2_Facet_Sphere_Dem3DofGeom**, is called for interaction of **Facet** with **Sphere** and might create (again) a **Dem3DofGeom** instance.

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

IPhysDispatcher dispatches to the second set of functors based on combination of 2 **Materials**; these functors 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**.

LawDispatcher 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.

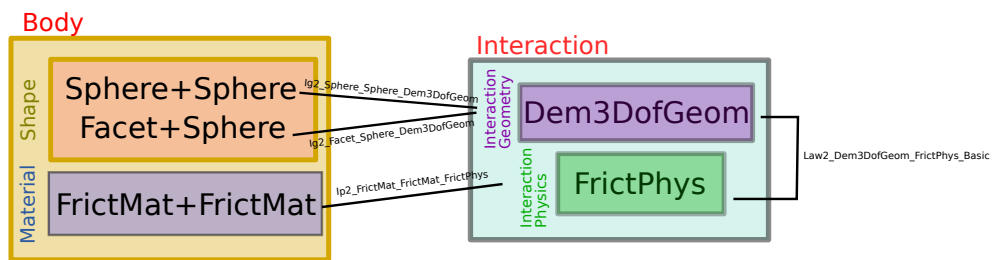


Figure 1.2: Chain of functors producing and accepting certain types. In the case shown, the **Ig2** functors produce **Dem3DofGeom** instances from all handled **Shape** combinations; the **Ig2** functor produces **FrictMat**. The constitutive law functor **Law2** accepts the combination of types produced. Note that the types are stated in the functor’s class names.

Chapter 2

User's manual

2.1 Scene construction

2.1.1 Triangulated surfaces

Yade integrates with the [GNU Triangulated Surface library](#), exposed in python via the 3rd party `gts` module. GTS provides variety of functions for surface manipulation (coarsening, tessellation, 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 than `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).

2.1.2 Sphere packings

Representing a solid of an arbitrary shape by arrangement of spheres presents the problem of sphere packing, i.e. spatial arrangement of sphere such that 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: $\text{mean} \pm \text{dispersion}$; 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-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)

```

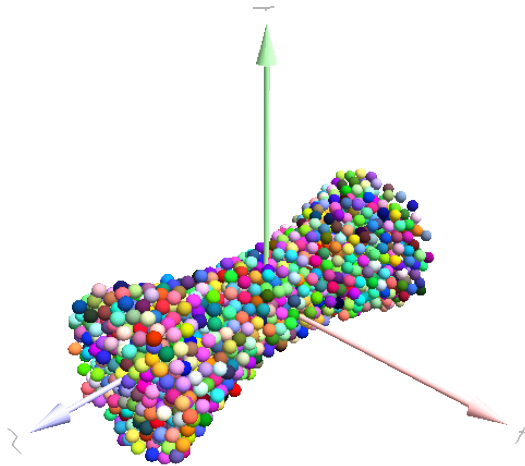


Figure 2.1: Specimen constructed with the `pack.inHyperboloid` predicate, packed with `pack.randomDensePack`.

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

# 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 pi
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))
```

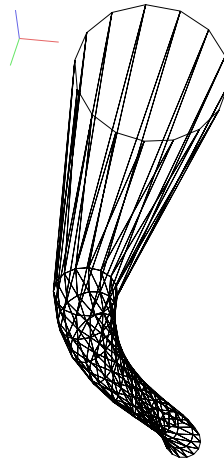


Figure 2.2: Triangulated funnel, constructed with the `examples/funnel.py` script.

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 `scripts/test/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)  
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))
```

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`):

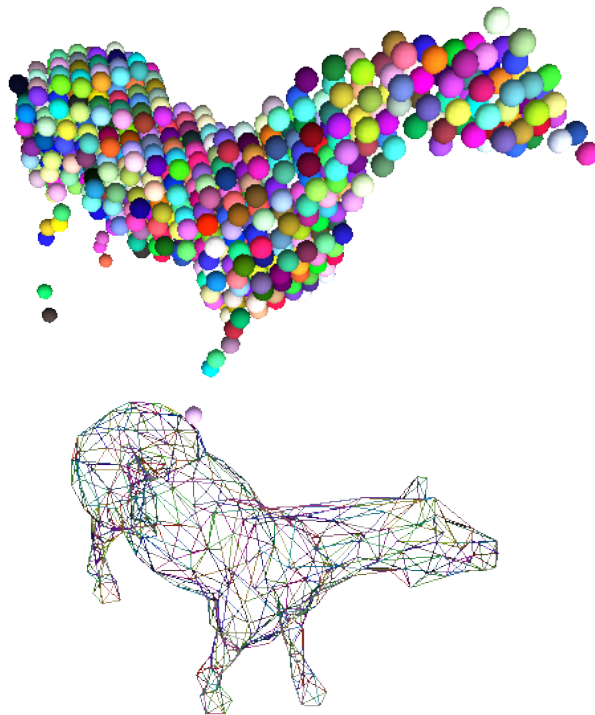


Figure 2.3: Imported GTS surface (horse) used as packing predicate (top) and surface constructed from facets (bottom). See <http://www.youtube.com/watch?v=PZVruIlUX1A> for movie of this simulation.

```
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)
```

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. `utils.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 `utils.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 `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)
```

Keyword arguments `wire`, `color` and `material` are not declared in `pack.randomDensePack`, therefore will be passed to `utils.sphere`, where they are also documented. `spheres` is now list of `Body` objects, which we add to the simulation:

```
O.bodies.append(spheres)
```

Packing algorithms described below produce dense packings. If one needs loose packing, `pack.SpherePack` class provides functions for generating loose packing, via its `pack.SpherePack.makeCloud` method. It is used internally for generating initial configuration in dynamic algorithms. For instance:

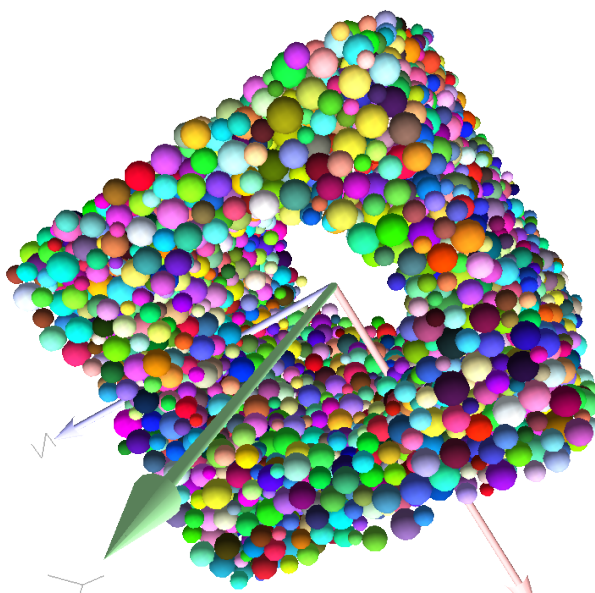


Figure 2.4: Box with cylinder removed from inside, using difference of these two predicates.

```
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:

```
for c,r in sp: O.bodies.append(utils.sphere(c,r))
```

or, in a more pythonic way, with one single `O.bodies.append` call:

```
O.bodies.append([utils.sphere(c,r) for c,r in sp])
```

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.

pack.SpherePadder constructs dense sphere packing based on pre-computed tetrahedron mesh; it is documented in [pack.SpherePadder](#) documentation; sample script is in [scripts/test/SpherePadder.py](#). **pack.SpherePadder** does not return **Body** list as other algorithms, but a **pack.SpherePack** object; it can be iterated over, adding spheres to the simulation, as shown in its documentation.

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 [scripts/test/genCylLSM.py](#). Full documentation for GenGeo can be found at [ESyS documentation website](#).

To our knowledge, the GenGeo library is not currently packaged. It can be downloaded from current subversion repository

```
svn checkout https://svn.esscc.uq.edu.au/svn/esys3/lsm/contrib/LSMGenGeo
```

then following instruction in the **INSTALL** file.

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.

2.1.3 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 `utils.sphere`, `utils.facet` and `utils.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 instance (though there is no limit).

`label` given to each material is optional, but can be passed to `utils.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 `utils.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 `utils.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 using `utils.defaultMaterial`; this should not happen for serious simulations, but is handy in simple scripts, where exact material properties are more or less irrelevant.

```
Yade [214]: len(O.materials)
-> [214]: 0

Yade [215]: idConcrete=O.materials.append(FrictMat(young=30e9,poisson=.2,frictionAngle=.6,label="concrete"))

Yade [216]: O.materials[idConcrete]
-> [216]: <FrictMat instance at 0x71ea020>

# uses the last defined material
Yade [218]: O.bodies.append(utils.sphere(center=(0,0,0),radius=1))
-> [218]: 0

# material given by id
Yade [220]: O.bodies.append(utils.sphere((0,0,2),1,material=idConcrete))
-> [220]: 1

# material given by label
Yade [222]: O.bodies.append(utils.sphere((0,2,0),1,material="concrete"))
-> [222]: 2

Yade [223]: idSteel=O.materials.append(FrictMat(young=210e9,poisson=.25,frictionAngle=.8,label="steel"))

Yade [224]: len(O.materials)
-> [224]: 2

# implicitly uses "steel" material, as it is the last one now
Yade [226]: O.bodies.append(utils.facet([(1,0,0),(0,1,0),(-1,-1,0)]))
-> [226]: 3
```

Adding multiple particles

As shown above, bodies are added one by one or several at the same time using the `append` method:

```
Yade [228]: O.bodies.append(utils.sphere((0,0,0),1))
-> [228]: 0

Yade [229]: O.bodies.append(utils.sphere((0,0,2),1))
```



```

-> [229]: 1

# this is the same, but in one function call
Yade [231]: O.bodies.append([
.....:     utils.sphere((0,0,0),1),
.....:     utils.sphere((0,0,2),1)
.....: ])
-> [234]: [2, 3]

```

Many functions introduced in preceding 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 `utils.sphere` and `utils.facet` internally, they accept additional argument passed to those function. 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); a special function `BodyContainer.appendClumped` is designed for this task; for instance, we might add 2 spheres tied together:

```

Yade [236]: O.bodies.appendClumped([
.....:     utils.sphere([0,0,0],1),
.....:     utils.sphere([0,0,2],1)
.....: ])
-> [239]: (2, [0, 1])

Yade [240]: len(O.bodies)
-> [240]: 3

Yade [241]: O.bodies[1].isClumpMember, O.bodies[2].clumpId
-> [241]: (True, 2)

Yade [242]: O.bodies[2].isClump, O.bodies[2].clumpId
-> [242]: (True, 2)

```

`appendClumped` returns a tuple of `(clumpId, [memberId1, memberId2])`: clump is internally represented by a special `Body`, referenced by `clumpId` of its members (see also `isClump`, `isClumpMember` and `isStandalone`).

2.1.4 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. (Either `Ig2_Sphere_Sphere_Dem3DofGeom.distFactor` or `Ig2_Sphere_Sphere_ScGeom.interactionDetectionFactor`, depending on the functor that is in use.

Note that only `Sphere + Sphere` interactions are supported; there is no parameter analogous to `distFactor` in `Ig2_Facet_Sphere_Dem3DofGeom`. 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

O.engines=[
    ForceResetter(),
    InsertionSortCollider([
        # enlarge here
        Bo1_Sphere_Aabb(aabbEnlargeFactor=intRadius,label='bo1s'),
        Bo1_Facet_Aabb(),
    ]),
    InteractionLoop(
        [
            # enlarge here
            Ig2_Sphere_Sphere_Dem3DofGeom(distFactor=intRadius,label='ig2ss'),
            Ig2_Facet_Sphere_Dem3DofGeom(),
        ],
        [Ip2_CpmMat_CpmMat_CpmPhys()],
        [Law2_Dem3DofGeom_CpmPhys_Cpm(epsSoft=0)], # deactivated
    ),
    NewtonIntegrator(damping=damping,label='damper'),
]

# run one single step
O.step()

# reset interaction radius to the default value
# see documentation of those attributes for the meaning of negative values
bo1s.aabbEnlargeFactor=-1
ig2ss.distFactor=-1

# 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 `utils.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 [244]: O.materials.append(FrictMat(young=3e10,poisson=.2,density=1000))
-> [244]: 0
```

```
Yade [245]: O.bodies.append([
.....:     utils.sphere([0,0,0],1),
.....:     utils.sphere([0,0,1000],1)
.....: ])
```

```

-> [248]: [0, 1]

# only add InteractionLoop, no other engines are needed now
Yade [249]: O.engines=[
.....:     InteractionLoop(
.....:         [Ig2_Sphere_Sphere_Dem3DofGeom()],
.....:         [Ip2_FrictMat_FrictMat_FrictPhys()],
.....:         [] # not needed now
.....:     )
.....: ]

Yade [256]: i=utils.createInteraction(0,1)

# created by functors in InteractionLoop
Yade [257]: i.geom, i.phys
-> [257]:
(<Dem3DofGeom_SphereSphere instance at 0x81bf680>,
 <FrictPhys instance at 0x82fb1a0>)

```

This method will be rather slow if many interaction are to be created (the functor lookup will be repeated for each of them). In such case, ask on yade-dev@lists.launchpad.net to have the `utils.createInteraction` function accept list of pairs id's as well.

2.1.5 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 apply 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:

```
0.engines=0.engines[:2]+[SomeEngine()]+0.engines[2:]
```

Functors choice

In the above example, we omitted 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 (September 2010), 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:

```
0.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 the derive from `IGeomFunctor`) depends on

1. shape combinations that should collide; for instance:

```
InteractionLoop([Ig2_Sphere_Sphere_Dem3DofGeom()],[],[])
```

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_Dem3DofGeom(),
    Ig2_Facet_Sphere_Dem3DofGeom()
],[],[])
```

Again, missing combination will cause given shape combinations to freely interpenetrate one another.

2. `IGeom` type accepted by the `Law2` functor (below); it is the first part of functor's name after `Law2` (for instance, `Law2_Dem3DofGeom_CpmPhys_Cpm` accepts `Dem3DofGeom`). This is (for most cases) either `Dem3DofGeom` (total shear formulation) or `ScGeom` (incremental shear formulation). For `ScGeom`, the above example would simply change to:

```
InteractionLoop([
    Ig2_Sphere_Sphere_ScGeom(),
    Ig2_Facet_Sphere_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 `Ig2` 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.

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_Dem3DofGeom_CpmPhys_Cpm` constitutive law.

- We use `spheres` and `facets` in the simulation, which selects `Ig2` functors accepting those types and producing `Dem3DofGeom`: `Ig2_Sphere_Sphere_Dem3DofGeom` and `Ig2_Facet_Sphere_Dem3DofGeom`.
- 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_Dem3DofGeom(), Ig2_Facet_Sphere_Dem3DofGeom()],  
    [Ip2_CpmMat_CpmMat_CpmPhys()],  
    [Law2_Dem3DofGeom_CpmPhys_Cpm()]  
)
```

2.1.6 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.

Motion constraints

- `Body.dynamic` determines whether a body will be moved by `NewtonIntegrator`; it is mandatory for bodies with zero mass, where applying non-zero force would result in infinite displacement.

`Facets` are case in the point: `utils.facet` makes them non-dynamic by default, as they have zero volume and zero mass (this can be changed, by passing `dynamic=True` to `utils.facet` or setting `Body.dynamic`; setting `State.mass` to a non-zero value must be done as well). The same is true for `utils.wall`.

Making sphere non-dynamic is achieved simply by:

```
utils.sphere([x,y,z],radius,dynamic=False)
```

Note: There is an open [bug #398089](#) to define exactly what the `dynamic` flag does. Please read it before writing a new engine relying on this flag.

- `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 [259]: O.bodies.append(utils.sphere((0,0,0),1))  
-> [259]: 0  
  
Yade [260]: O.bodies[0].state.blockedDOFs=['z','rx','ry']
```

In contrast to `Body.dynamic`, `blockedDOFs` will only block forces (and acceleration) in that direction being effective; if you prescribed linear or angular velocity, they will be applied regardless of `blockedDOFs`. (This is also related to [bug #398089](#) mentioned above)

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 exploited here as well:

```
from yade import pack  
pred=pack.inAlignedBox(lowerCorner,upperCorner)  
for b in O.bodies:  
    if b.shape.name!=Sphere: continue # skip non-spheres
```

```
# ask the predicate if we are inside
if pred(b.state.pos, b.shape.radius): b.dynamic=False
```

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) ([ThreeDTriaxialEngine](#) is generalized such that it allows independent value of stress along each axis)
- [PeriTriaxController](#) for applying stress/strain along 3 axes independently, for simulations using periodic boundary conditions ([Cell](#))

Field appliers

Engines deriving from [FieldApplier](#) acting on all particles. The one most used is [GravityEngine](#) applying uniform acceleration field.

Partial engines

Engines deriving from [PartialEngine](#) define the [subscribedBodies](#) 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.

If you need an engine applying non-constant value instead, there are several interpolating engines ([InterpolatingDirectedForceEngine](#) for applying force with varying magnitude, [InterpolatingSpiralEngine](#) for applying spiral displacement with varying angular velocity and possibly others); writing a new interpolating engine is rather simple using examples of those that already exist.

2.1.7 Convenience features

Labeling things

Engines and functors can define that `label` attribute. Whenever the `0.engines` sequence is modified, python variables of those names are created/update; 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 [262]: dict(O.tags) # convert to real dictionary
-> [262]:
{'author': '~(bchareyre@dt-rv020)',
 'd.id': '20110211T183255p6478',
 'id': '20110211T183255p6478',
 'id.d': '20110211T183255p6478',
 'isoTime': '20110211T183255'}
```

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 *batch-output-separate* 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 [263]: O.tags['anythingThat I lik3']='whatever'
```

```
Yade [264]: O.tags['anythingThat I lik3']
-> [264]: '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 `utils.saveVars` and `utils.loadVars`.

`utils.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 `utils.loadVars` with the same identification *mark*:

```
Yade [265]: a=45; b=pi/3

Yade [266]: utils.saveVars('ab',a=a,b=b)

# save simulation (we could save to disk just as well)
Yade [266]: O.saveTmp()

Yade [268]: O.loadTmp()

Yade [269]: utils.loadVars('ab')

Yade [270]: yade.params.ab.a
-> [270]: 45
```



```

# import like this
Yade [271]: from yade.params import ab

Yade [272]: ab.a, ab.b
-> [272]: (45, 1.0471975511965976)

# also possible
Yade [273]: from yade.params import *

Yade [274]: ab.a, ab.b
-> [274]: (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=a*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)
    #
    utils.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 `utils.saveVars`.

2.2 Controlling simulation

2.2.1 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 `utils.kineticEnergy`) every 5 seconds, this engine will be put `PyRunner(command="print 'kinetic energy',utils.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 id1,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)
]

```

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 [276]: from yade import plot
```

```
Yade [277]: plot.data
```

```
-> [277]:
```

```
{'eps': [0.0001, 0.001, nan, 0.0001, 0.001, nan, 0.0001, 0.001, nan],  
  'force': [nan, nan, 1000.0, nan, nan, 1000.0, nan, nan, 1000.0],  
  'sigma': [12, nan, nan, 12, nan, nan, 12, nan, nan]}
```

```
Yade [278]: 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 [279]: plot.addData(eps=1e-3)
```

```
# adds NaNs to already existing sigma and eps columns
```

```
Yade [280]: plot.addData(force=1e3)
```

```
Yade [281]: plot.data
```

```
-> [281]:
```

```
{'eps': [0.0001,  
         0.001,  
         nan,  
         0.0001,  
         0.001,  
         nan,  
         0.0001,  
         0.001,  
         nan,  
         0.0001,  
         0.001,  
         nan],  
  'force': [nan,  
            nan,  
            1000.0,  
            nan,  
            nan,  
            1000.0,  
            nan,  
            nan,  
            1000.0,  
            nan,  
            nan,  
            1000.0],  
  'sigma': [12, nan, nan, 12, nan, nan, 12, nan, nan]}
```

```
# retrieve only one column
```

```
Yade [282]: plot.data['eps']
```

```
-> [282]:
```

```
[0.0001,  
 0.001,  
 nan,  
 0.0001,  
 0.001,  
 nan,  
 0.0001,  
 0.001,  
 nan]
```

```
nan,
0.0001,
0.001,
nan]
```

```
# get maximum eps
Yade [283]: max(plot.data['eps'])
-> [283]: 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 column label for `plot.data`. If it cannot be used as keyword name for `plot.addData` (since it is a python keyword (`for`), or has spaces inside (`my funny column`), you can pass dictionary to `plot.addData` instead:

```
plot.addData(z=b.state.pos[2],**{'my funny column':b.state.vel.norm()})
```

An exception are columns having leading or 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.stress,eps=strainer.strain)
```

In that case, naturally, the labeled object must define attributes which are used (`UniaxialStrainer.strain` and `UniaxialStrainer.stress` 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 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')}
```

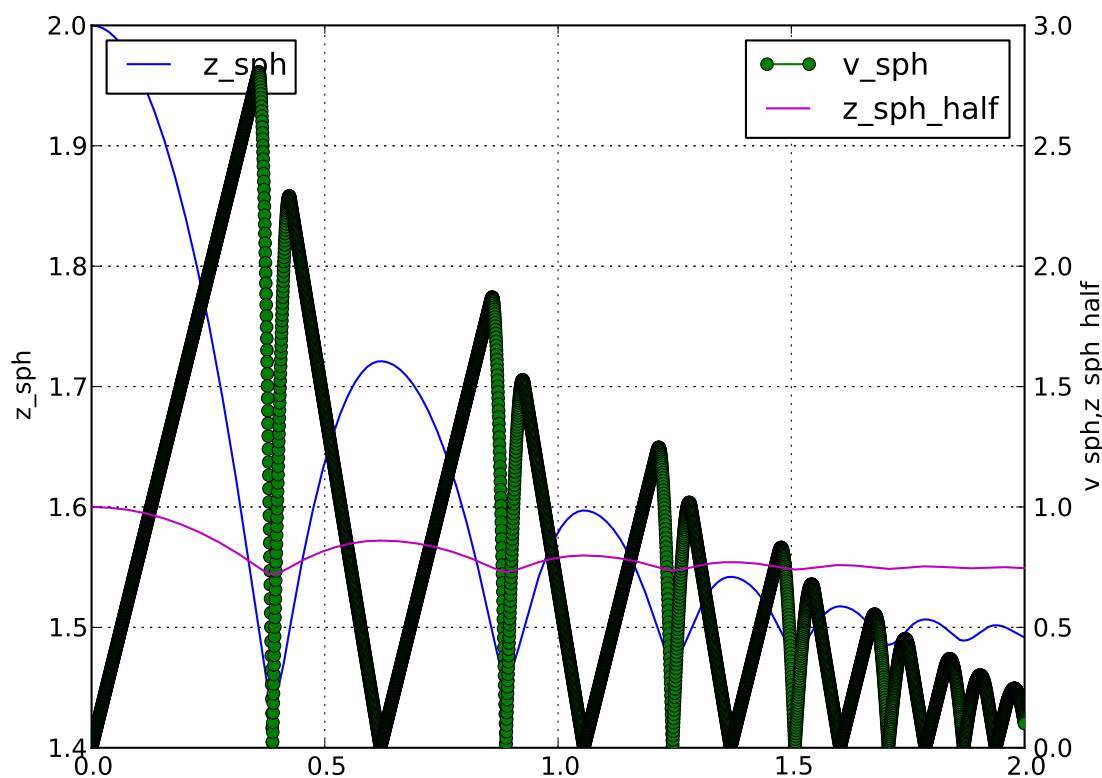


Figure 2.5: Figure generated by `examples/simple-scene/simple-scene-plot.py`.

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-.'))}
```

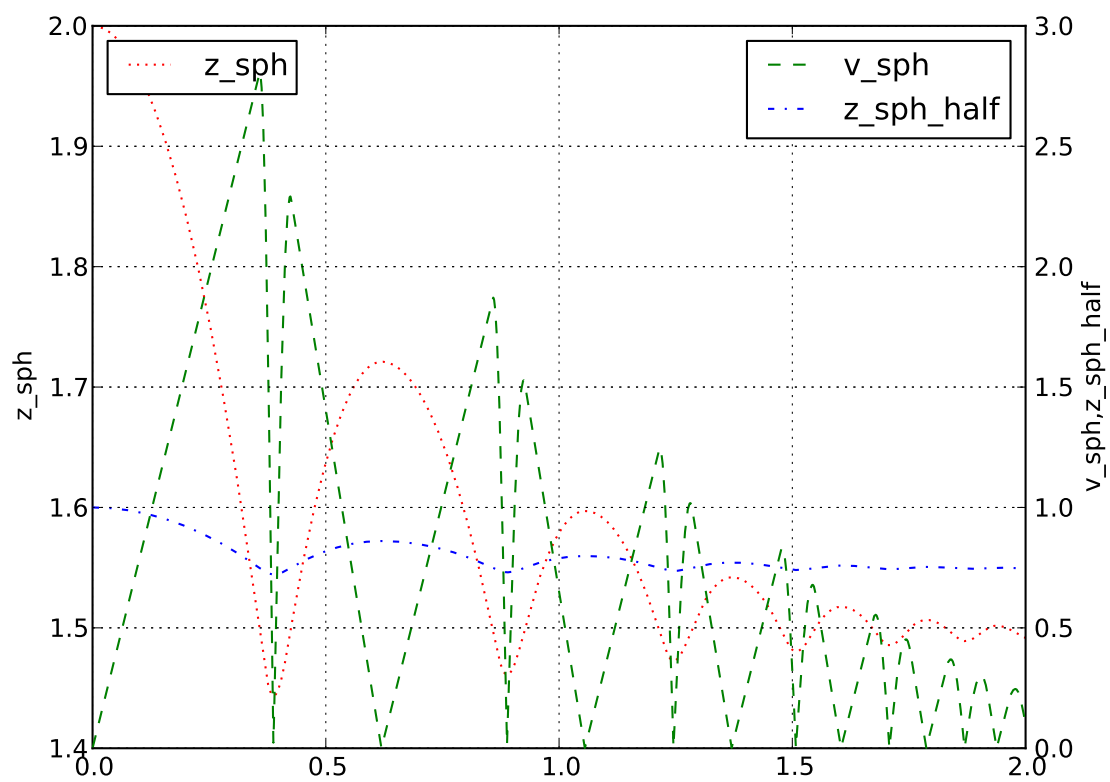


Figure 2.6: Figure generated by changing parameters to `plot.plots` as above.

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-.'))}
```

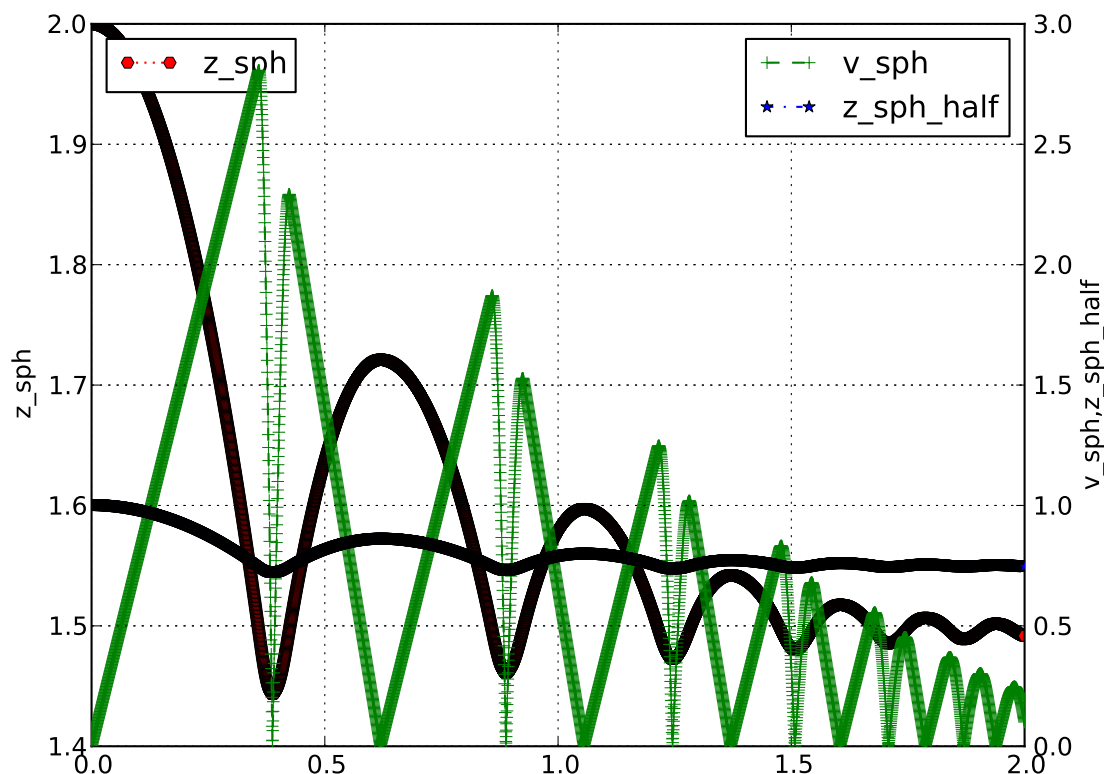


Figure 2.7: Figure generated by changing parameters to `plot.plots` as above.

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-.'))}
plot.labels={'z_sph':'$z_{sph}$' , 'v_sph':'$v_{sph}$' , 'z_sph_half':'$z_{sph}/2$'}
```

Greek letters are simply a `'α'`, `'β'` 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>

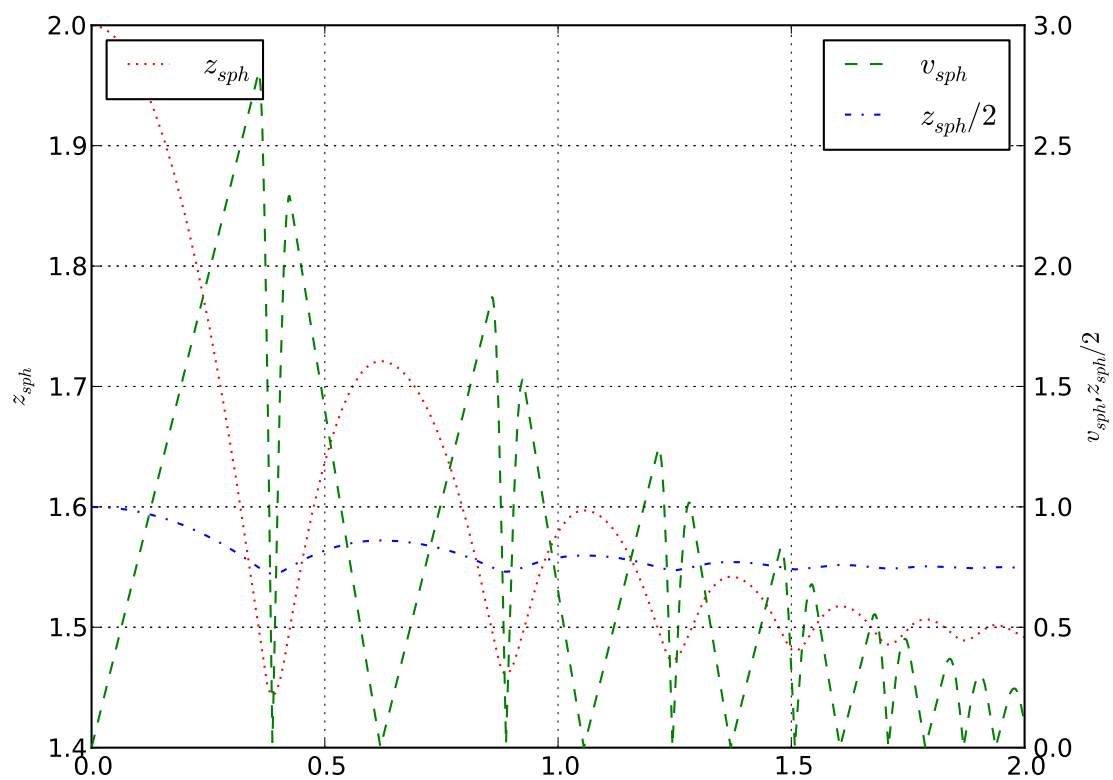


Figure 2.8: Figure generated by `examples/simple-scene/simple-scene-plot.py`, with TeX labels.

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 [284]: plot.plots={
.....:     'i':('t',),
.....:     'i':('z1','v1')
.....: }
```

```
Yade [288]: plot.plots
-> [288]: {'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 [289]: plot.plots={
.....:     'i':('t',),
.....:     'i ':('z1','v1') # note the space in 'i '
.....: }
```

```
Yade [293]: plot.plots
-> [293]: {'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 can be exported to external files for later post-processing via that `plot.saveGnuplot` function.

- 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.

2.2.2 Stop conditions

For simulations with pre-determined number of steps, number of steps 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 `_utils.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 utils.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 utils.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()
```

2.2.3 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:

```
$ 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 are only allowed from localhost**, not over network remotely.

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-multi.in` for details.

2.2.4 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 `utils.readParamsFromTable` to obtain parameters from parameter table. In order to make the script runnable outside the batch, `utils.readParamsFromTable` takes default values of parameters, which might be overridden from the parameter table.

`utils.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 `utils.readParamsFromTable` (using `utils.TableParamReader` class), called from simulation script, as explained above.

The batch can be run as

```
yade-batch parameters.table simulation.py
```

and it will intelligently run one simulation for each parameter table line.

Example

This example is found in `scripts/batch.table` and `scripts/batch.py`.

Suppoe 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_rho_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:

```
from yade import utils
utils.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 `utils.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
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_rho_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 --threads=4 --nice=10 -x bat
#1 (hi_v/4): PARAM_TABLE=batch.table:3 DISPLAY= /usr/local/bin/yade-trunk --threads=4 --nice=10 -x batch.py
#2 (lo_v/4): PARAM_TABLE=batch.table:4 DISPLAY= /usr/local/bin/yade-trunk --threads=4 --nice=10 -x batch.py
#3 (hi_rho/4): PARAM_TABLE=batch.table:5 DISPLAY= /usr/local/bin/yade-trunk --threads=4 --nice=10 -x batch.
#4 (hi_rho_v/4): PARAM_TABLE=batch.table:6 DISPLAY= /usr/local/bin/yade-trunk --threads=4 --nice=10 -x batch
#5 (hi_rho_lo_v/4): PARAM_TABLE=batch.table:7 DISPLAY= /usr/local/bin/yade-trunk --threads=4 --nice=10 -x b
```

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_rho_lo_v/4) started on Tue Apr 13 13:59:40 2010
#5 (hi_rho_lo_v/4) done (exit status 0), duration 00:00:01, log batch.hi_rho_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_rho_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.

For smaller simulations, prepending all output file names with `O.tags['id']` can be sufficient:

```
utils.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'),
    # ...
]
# ...
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, 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.

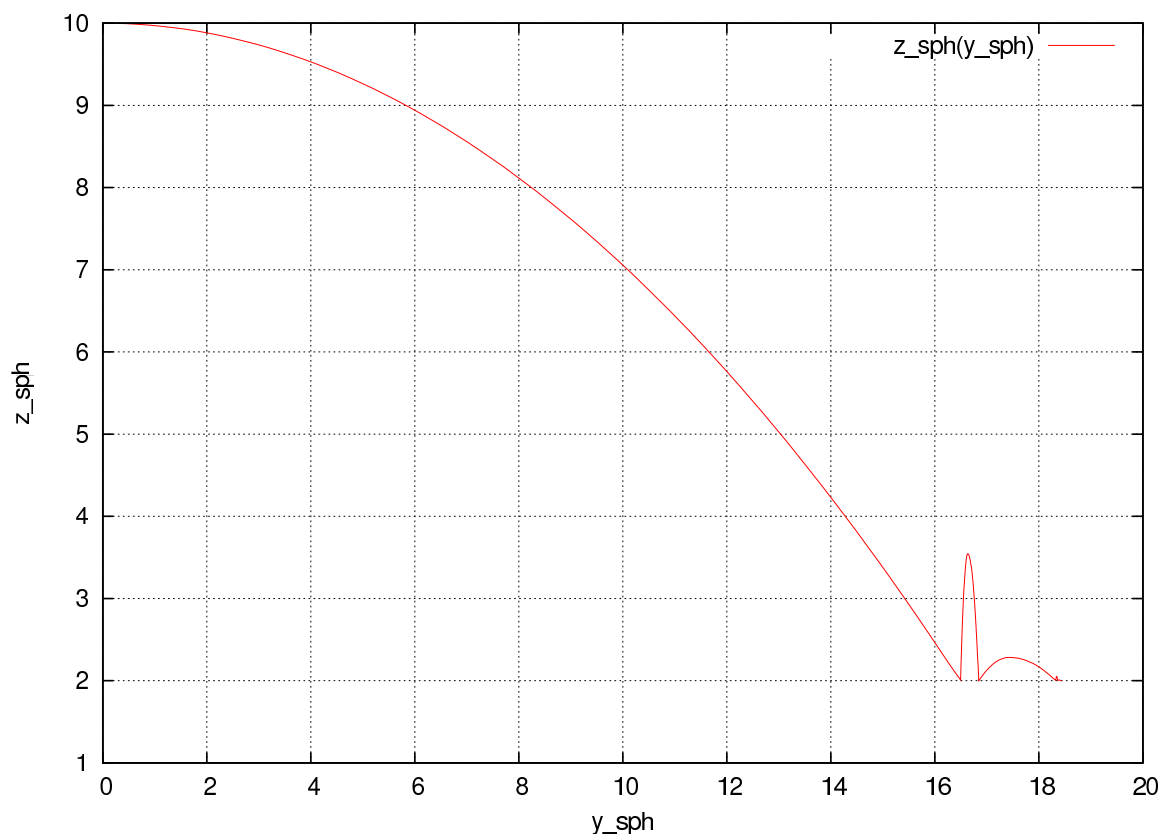


Figure 2.9: Figure from single job in the batch.

HTTP overview

While job is running, the batch system presents progress via simple HTTP server running at port 9080, which can be accessed from regular web browser by requesting the `http://localhost:9080` URL. This page can be accessed remotely over network as well.

2.3 Postprocessing

2.3.1 3d rendering & videos

There are multiple ways to produce a video of simulation:

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/bulldozer.py](#) for a full example):

```
0.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 `utils.encodeVideoFromFrames`:

```
utils.encodeVideoFromFrames(snapshooter.savedSnapshots,out='/tmp/bulldozer.ogv',fps=2)
```

The video is encoded in the [Theora](#) format stored in an ogg container.

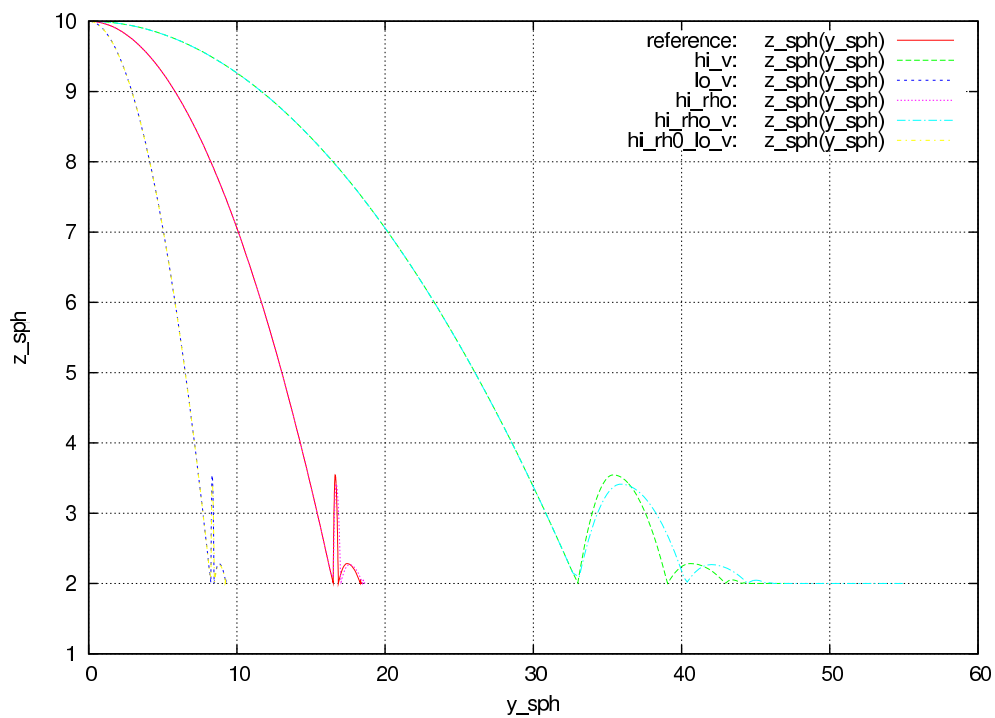


Figure 2.10: Merged figure from all jobs in the batch. Note that labels are prepended by job description to make lines distinguishable.

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

Figure 2.11: Summary page available at port 9080 as batch is processed (updates every 5 seconds automatically). Possible job statuses are pending, running, done, failed.

3. Specialized post-processing tools, notably [Paraview](#). This is described in more detail in the following section.

Paraview

Saving data during the simulation

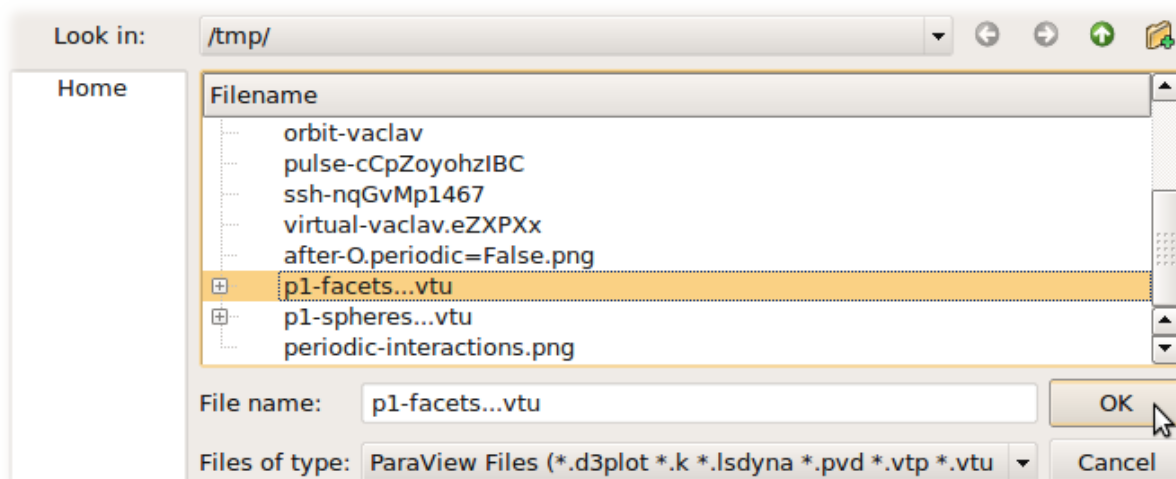
Paraview is based on the [Visualization Toolkit](#), which defines formats for saving various types of data. One of them (with the `.vtu` extension) can be written by a special engine [VTKRecorder](#). It is added to the simulation loop:

```
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 (see the [documentation](#))


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:

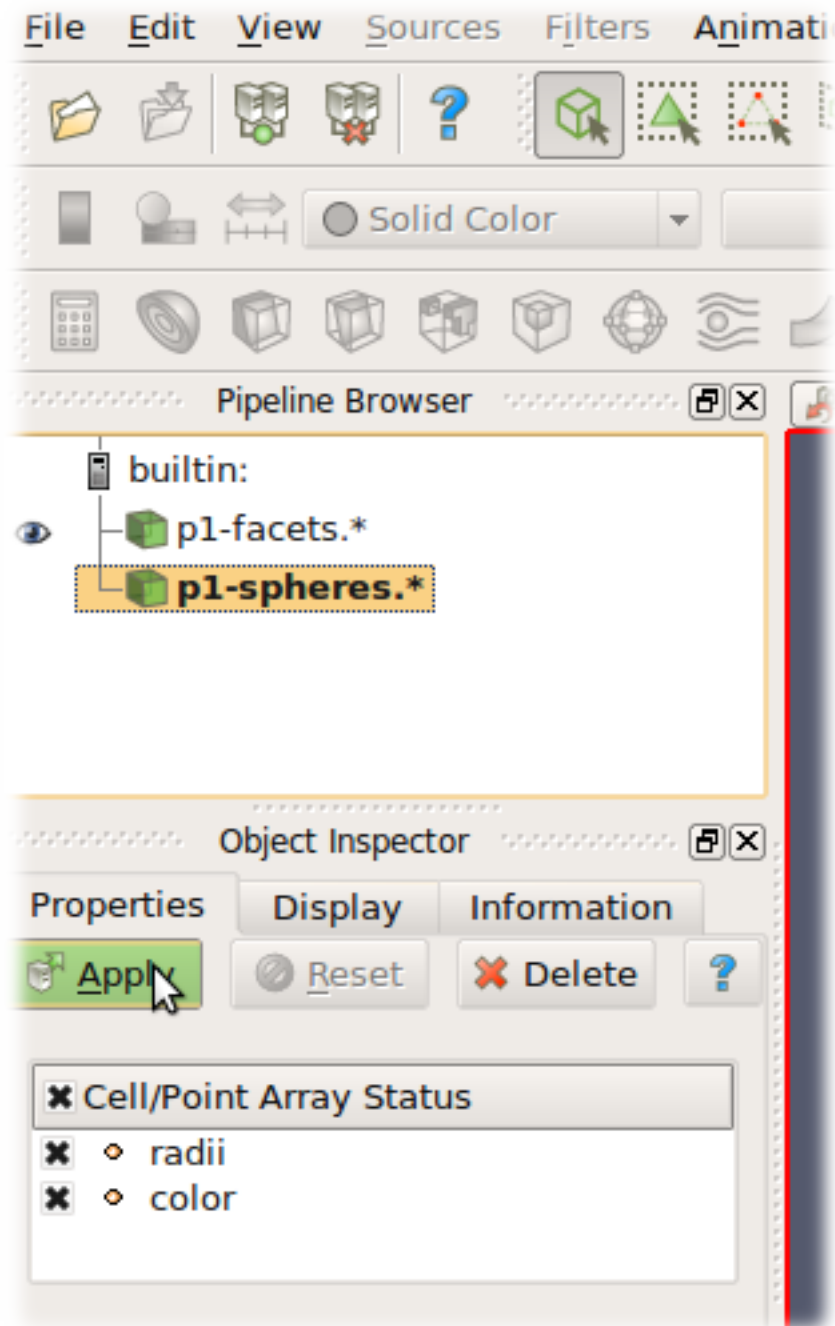


Click on the “Apply” button in the “Object inspector” sub-window to make loaded objects visible. You can see tree of displayed objects in the “Pipeline browser”:

Rendering spherical particles 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.

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*.

2.4 Python specialties and tricks

2.5 Extending Yade

- new particle shape
- new constitutive law

2.6 Troubleshooting

2.6.1 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 you 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/u
#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/pos
#7  0x000000000004a7c5e in call_function (f=Frame 0x1c54620, for file <ipython console>, line 1, in <module> ()),
#8  PyEval_EvalFrameEx (f=Frame 0x1c54620, for file <ipython console>, line 1, in <module> ()), throwflag=<value
```

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.

2.6.2 Reporting bugs

Bugs are general name for defects (functionality shortcomings, misdocumentation, crashes) or feature requests. They are tracked at <http://bugs.launchpad.net/yade>.

When reporting a new bug, be as specific as possible; state version of yade you use, system version and so on, as explained in the above section on crashes.

2.6.3 Getting help

Mailing lists

Yade has two mailing-lists. Both are hosted at <http://www.launchpad.net> and before posting, you must register to Launchpad and subscribe to the list by adding yourself to “team” of the same name running the list.

yade-users@lists.launchpad.net is general help list for Yade users. Add yourself to yade-users team so that you can post messages. [List archive](#) is available.

yade-dev@lists.launchpad.net is for discussions about Yade development; you must be member of yade-dev team to post. This list is [archived](#) as well.

Read [How To Ask Questions The Smart Way](#) before posting. Do not forget to state what *version* of yade you use (shown when you start yade), what operating system (such as Ubuntu 10.04), and if you have done any local modifications to source code.

Questions and answers

Launchpad provides interface for giving questions at <https://answers.launchpad.net/yade/> which you can use instead of mailing lists; at the moment, it functionality somewhat overlaps with yade-users, but has the advantage of tracking whether a particular question has already been answered.

Wiki

<http://www.yade-dem.org/wiki/>

Private and/or paid support

You might contact developers by their private mail (rather than by mailing list) if you do not want to disclose details on the mailing list. 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.

Chapter 3

Programmer's manual

3.1 Build system

Yade uses `scons` build system for managing the build process. It takes care of configuration, compilation and installation. SCons is written in python and its build scripts are in python, too. SCons complete documentation can be found in its manual page.

3.1.1 Pre-build configuration

We use `$` to denote build variable in strings in this section; in SCons script, they can be used either by writing `$variable` in strings passed to SCons functions, or obtained as attribute of the `Environment` instance `env`, i.e. `env['variable']`; we use the formed in running text here.

In order to allow parallel installation of multiple yade versions, the installation location follows the pattern `$PREFIX/lib/yade$SUFFIX` for libraries and `$PREFIX/bin/yade$SUFFIX` for executables (in the following, we will refer only to the first one). `$SUFFIX` takes the form `-$version$variant`, which further allows multiple different builds of the same version (typically, optimized and debug builds). For instance, the default debug build of version 0.5 would be installed in `/usr/local/lib/yade-0.5-dbg/`, the executable being `/usr/local/bin/yade-0.5-dbg`.

The build process takes place outside the source tree, in directory referred to as `$buildDir` within those scripts. By default, this directory is `../build-$SUFFIX`.

Each build depends on a number of configuration parameters, which are stored in mutually independent *profiles*. They are selected according to the `profile` argument to `scons` (by default, the last profile used, stored in `scons.current-profile`). Each profile remembers its non-default variables in `scons.profile-$profile`.

There is a number of configuration parameters; you can list all of them by `scons -h`. The following table summarizes only a few that are the most used.

PREFIX [default: `/usr/local`] installation prefix (PREFIX preprocessor macro; `yade.config.prefix` in python)

version [`bzr revision (e.g. bzr1899)`] first part of suffix (SUFFIX preprocessor macro; `yade.config.suffix` in python)

variant [*(empty)*] second part of suffix

buildPrefix [`..`] where to create `build-$SUFFIX` directory

debug [`False (0)`] add debugging symbols to output, enable stack traces on crash

optimize [`True (1)`] optimize binaries (`#define NDEBUG`; assertions eliminated; `YADE_CAST` and `YADE_PTR_CAST` are static casts rather than dynamic; `LOG_TRACE` and `LOG_DEBUG` are eliminated)

CPPPATH [`/usr/include/vtk-5.2:/usr/include/vtk-5.4`] additional colon-separated paths for pre-processor (for atypical header locations). Required by some libraries, such as VTK (reflected by the default)

LIBPATH [*(empty)*] additional colon-separated paths for linker

CXX [`g++`] compiler executable

CXXFLAGS [*(empty)*] additional compiler flags (may be added automatically)

jobs [`4`] number of concurrent compilations to run

brief [*True* (`1`)] only show brief notices about what is being done rather than full command-lines during compilation

linkStrategy [`monolithic`] whether to link all plugins in one shared library (`monolithic`) or in one file per plugin (`per-class`); the first option is faster for overall builds, while the latter one makes recompilation of only part of Yade faster; granularity of monolithic build can be changed with the `chunkSize` parameter, which determines how many files are compiled at once.

features [`log4cxx,opengl,gts,openmp`] optional comma-separated features to build with (details below; each defines macro `YADE_$FEATURE`; available as lowercased list `yade.config.features` at runtime)

Library detection

When the `scons` command is run, it first checks for presence of all required libraries. Some of them are *essential*, other are *optional* and will be required only if features that need them are enabled.

Essentials

compiler Obviously `c++` compiler is necessary. Yade relies on several extensions of `g++` from the ‘`gcc`’ <<http://gcc.gnu.org>> suite and cannot (probably) be built with other compilers.

boost `boost` is a large collection of peer-reviewed `c++` libraries. Yade currently uses `thread`, `date_time`, `filesystem`, `iostreams`, `regex`, `serialization`, `program_options`, `foreach`, `python`; typically the whole `boost` bundle will be installed. If you need functionality from other modules, you can make presence of that module mandatory. Only be careful about relying on very new features; due to range of systems yade is or might be used on, it is better to be moderately conservative (read: roughly 3 years backwards compatibility).

python ‘`python`’ <<http://www.python.org>> is the scripting language used by yade. Besides [`boost::python`], yade further requires

- `ipython` (terminal interaction)
- `matplotlib` (plotting)
- ‘`numpy`’ <<http://www.numpy.org>> (matlab-like numerical functionality and accessing numpy arrays from `c/c++` efficiently)

Optional libraries (features)

The *features* parameter controls optional functionality. Each enabled feature defines preprocessor macro `YADE_FEATURE` (name uppercased) to enable selective exclude/include of parts of code. Code of which compilation depends on a particular features should use `#ifdef YADE_FEATURE` constructs to exclude dependent parts.

log4cxx (`YADE_LOG4CXX`) Enable flexible logging system (`log4cxx`), which permits to assign logging levels on per-class basis; doesn’t change API, only redefines `LOG_INFO` and other macros accordingly; see *log4cxx* for details.

opengl (`YADE_OPENGL`) Enable 3d rendering as well as the Qt3-based graphical user interface (in addition to `python` console).

vtk (**YADE_VTK**) Enable functionality using Visualization Toolkit (**vtk**; e.g. **VTKRecorder** exporting to files readable with ParaView).

openmp (**YADE_OPENMP**) Enable parallelization using OpenMP, non-intrusive shared-memory parallelization framework; it is only supported for **g++** > 4.0. Parallel computation leads to significant performance increase and should be enabled unless you have a special reason for not doing so (e.g. single-core machine). See *upyade-parallel* for details.

gts (**YADE_GTS**) Enable functionality provided by GNU Triangulated Surface library (**gts**) and build PyGTS, its python interface; used for surface import and construction.

cgal (**YADE_CGAL**) Enable functionality provided by Computation Geometry Algorithms Library (**cgal**); triangulation code in **MicroMacroAnalyser** and **PersistentTriangulationCollider** uses its routines.

other There might be more features added in the future. Always refer to **scons -h** output for possible values.

Warning: Due to a long-standing **bug** in **log4cxx**, using **log4cxx** will make yade crash at every exit. We work-around this partially by disabling the crash handler for regular exits, but process exit status will still be non-zero. The batch system (*yade-multi*) detects successful runs by looking at magic line “Yade: normal exit.” in the process’ standard output.

Before compilation, SCons will check for presence of libraries required by their respective features ¹. Failure will occur if a respective library isn’t found. To find out what went wrong, you can inspect `../build-$SUFFIX/config.log` file; it contains exact commands and their output for all performed checks.

Note: Features are not auto-detected on purpose; otherwise problem with library detection might build Yade without expected features, causing specifically problems for automatized builds.

3.1.2 Building

Yade source tree has the following structure (omiting **debian**, **doc**, **examples** and **scripts** which don’t participate in the build process); we shall call each top-level component *module*:

```
attic/      ## code that is not currently functional and might be removed unless resurrected
  lattice/  ## lattice and lattice-like models
  snow/     ## snow model (is really a DEM)
core/       ## core simulation building blocks
extra/      ## miscillanea
gui/        ## user interfaces
  qt3/      ## graphical user interface based on qt3 and OpenGL
  py/       ## python console interface (phased out)
lib/        ## support libraries, not specific to simulations
pkg/        ## simulation-specific files
  common/   ## generally useful classes
  dem/      ## classes for Discrete Element Method
py/         ## python modules
```

Each directory on the top of this hierarchy (except **pkg**, which is treated specially – see below) contains file **SConstruct**, determining what files to compile, how to link them together and where should they be installed. Within these script, a **scons** variable **env** (build **Environment**) contains all the configuration parameters, which are used to influence the build process; they can be either obtained with the `[]` operator, but **scons** also replaces **\$var** strings automatically in arguments to its functions:

```
if 'opengl' in env['features']:
    env.Install('$PREFIX/lib/yade$SUFFIX/', [
        # ...
    ])
```

¹ Library checks are defined inside the **SConstruct** file and you can add your own, should you need it.

Header installation

To allow flexibility in source layout, SCons will copy (symlink) all headers into flattened structure within the build directory. First 2 components of the original directory are joined by dash, deeper levels are discarded (in case of `core` and `extra`, only 1 level is used). The following table makes gives a few examples:

Original header location	Included as
core/Scene.hpp	<yade/core/Scene.hpp>
lib/base/Logging.hpp	<yade/lib-base/Logging.hpp>
lib/serialization/Serializable.hpp	<yade/lib-serialization/Serializable.hpp>
pkg/dem/DataClass/SpherePack.hpp	<yade/pkg-dem/SpherePack.hpp>
gui/qt3/QtGUI.hpp	<yade/gui-qt3/QtGUI.hpp>

It is advised to use `#include<yade/module/Class.hpp>` style of inclusion rather than `#include"Class.hpp` even if you are in the same directory.

What files to compile

SConscript files in `lib`, `core`, `gui`, `py` and `extra` explicitly determine what files will be built.

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* by SCons and considered for compilation. Each file may contain multiple lines that declare features that are necessary for this file to be compiled:

```
YADE_REQUIRE_FEATURE(vtk);
YADE_REQUIRE_FEATURE(gts);
```

This file will be compiled only if *both* `vtk` and `gts` features are enabled. Depending on current feature set, only selection of plugins will be compiled.

It is possible to disable compilation of a file by requiring any non-existent feature, such as:

```
YADE_REQUIRE_FEATURE(temporarily disabled 345uysdijkn);
```

The `YADE_REQUIRE_FEATURE` macro expands to nothing during actual compilation.

Note: The source scanner was written by hand and is not official part of SCons. It is fairly primitive and in particular, it doesn't interpret c preprocessor macros, except for a simple non-nested feature-checks like `#ifdef YADE_*/#ifndef YADE_* #endif`.

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 for monolithic builds, libClass.so for per-class (per-plugin) builds. (undefined)	core, yade-support; may not depend on external libraries explicitly (only implicitly, by adding the library to global linker flags in <code>SConstruct</code>) (arbitrary)
extra		
gui	libQtGUI.so, libPythonUI.so	lib, core, pkg
py	(many files)	lib, core, pkg, external

Because `pkg` plugins might be linked differently depending on the `linkStrategy` option, `SConscript` files that need to explicitly declare the dependency should use provided `linkPlugins` function which returns libraries in which given plugins will be defined:

```
env.SharedLibrary('_packSpheres',['_packSpheres.cpp'],
    SHLIBPREFIX='',
    LIBS=env['LIBS']+linkPlugins(['Shop','SpherePack']),
),
```

Note: `env['LIBS']` are libraries that all files are linked to and they should always be part of the `LIBS` parameter.

Since plugins in `pkg` are not declared in any `SConscript` file, other plugins they depend on are again found *automatically* by scanning their `#include` directives for the pattern `#include<yade/module/Plugin.hpp>`. Again, this works well in normal circumstances, but is not necessarily robust.

See `scons` manpage for meaning of parameters passed to build functions, such as `SHLIBPREFIX`.

3.2 Conventions

The following rules that should be respected; documentation is treated separately.

- general
 - C++ source files have `.hpp` and `.cpp` extensions (for headers and implementation, respectively).
 - All header files should have the `#pragma once` multiple-inclusion guard.
 - Try to avoid `using namespace ...` in header files.
 - 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 names (`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 use lowercased 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 logging abundantly. Again, `LOG_TRACE` and `LOG_DEBUG` are eliminated from optimized code; unless turned on explicitly, the output will be suppressed even in the debug build (see below).
 - 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 uninitialized variable. (This is taken care of by `YADE_CLASS_BASE_DOC_* macro family` macros for user classes)

3.2.1 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 ([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 dispatchers² are defined:

dispatcher	arity	dispatch types	created type	functor type	functor prefix
BoundDispatcher	1	Shape	Bound	BoundFunctor	Bo1
IGeomDispatcher	2 (symetric)	$2 \times$ Shape	IGeom	IGeomFunctor	Ig2
IPhysDispatcher	2 (symetric)	$2 \times$ Material	IPhys	IPhysFunctor	Ip2
LawDispatcher	2 (asymmetric)	IGeom IPhys	<i>(none)</i>	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_Dem3DofGeom](#) is binary functor called for [Facet](#) and [Sphere](#), creating and instace of [Dem3DofGeom](#).
- [Law2_Dem3DofGeom_CpmPhys_Cpm](#) is binary functor ([LawFunctor](#)) called for types [Dem3Dof \(Geom\)](#) and [CpmPhys](#).

3.2.2 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)

² Not considering OpenGL dispatchers, which might be replaced by regular virtual functions in the future.

- 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`.

Historically, Yade was using Doxygen for in-source documentation. This documentation is still available (by running `scons doc`), but was rarely written and used by programmers, and had all the disadvantages of auto-generated documentation. Then, as Python became ubiquitous in yade, python was documented using epydoc generator. Finally, hand-written documentation (this one) started to be written using [Sphinx](#), which was developed originally for documenting Python itself. Disadvantages of the original scatter were different syntaxes, impossibility for cross-linking, non-interactivity and frequently not being up-to-date.

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** <<http://docutils.sourceforge.net/rst.html>>’ (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.

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](#).

|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 update, 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 in Sphinx](#).

Bibliographical references

As in any scientific documentation, references to publications are very important. To cite an article, add it to BibTeX file in `doc/references.bib`, using the BibTeX format. 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 cited by `[Author2008]`; for example:

According to `[Allen1989]`, the integration scheme ...

will be rendered as

According to `[Allen1989]`, the intergration 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/_etraDocs.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).

Local documentation

Note: At some future point, this documentation will be integrated into yade’s sources. This section should be updated accordingly in that case.

To generate Yade’s documentation locally, get a copy of the `ydoc branch` via bzt, then follow instructions in the `README` file.

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.

3.3 Support framework

Besides the framework provided by the c++ standard library (including STL), boost and other dependencies, yade provides its own specific services.

3.3.1 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.

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.

3.3.2 Basic numerics

The floating point type to use in Yade `Real`, which is by default typedef for `double`.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`

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.

⁴ Historically, it was thought that Yade could be also run with single precision based on build-time parameter; it turned out however that the impact on numerical stability was such disastrous that this option is not available now. There is, however, `QUAD_PRECISION` parameter to `scons`, which will make `Real` a typedef for `long double` (extended precision; quad precision in the proper sense on IA64 processors); this option is experimental and is unlikely to be used in near future, though.

- 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 `miniEigen` 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.

```
# cross product
Yade [61]: Vector3(1,2,3).cross(Vector3(0,0,1))
-> [61]: Vector3(2,-1,0)

# construct quaternion from axis and angle
Yade [63]: Quaternion(Vector3(0,0,1),pi/2)
-> [63]: Quaternion((0,0,1),1.5707963267948966)
```

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.

3.3.3 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 getClass_name()` 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 [65]: yade.system.childClasses('Shape')
-> [65]:
set(['Box',
    'ChainedCylinder',
    'Clump',
    'Cylinder',
    'Facet',
    'Sphere',
    'Tetra',
    'Wall'])
```



```
Yade [66]: Sphere().name          ## getClassNames()
-----
AttributeError                    Traceback (most recent call last)

/home/3S-LAB/bchareyre/yade/yade-test-kdev/yade/doc/sphinx/<ipython console> in <module>()

AttributeError: 'Sphere' object has no attribute 'name'
```

3.3.4 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**).

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).

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 serialized 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 functionality 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,,"Doc
```

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 omitting **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 (It is possible that different mechanism will be in place if `boost::serialization` is used, though.)

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. 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 (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 attribute 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:

```
/*! 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 [kgms2]"))
);
};
// registration function for ClassFactory
REGISTER_SERIALIZABLE(GravityEngine);

```

and this is the implementation:

```

#include<yade/pkg-common/GravityEngine.hpp>
#include<yade/core/Scene.hpp>

// registering the plugin
YADE_PLUGIN((GravityEngine));

void GravityEngine::action(){
    /* do the work here */
}

```

We can create a mini-simulation (with only one GravityEngine):

```
Yade [67]: O.engines=[GravityEngine(gravity=Vector3(0,0,-9.81))]
```

```
Yade [68]: O.save('abc.xml')
```

and the XML looks like this:

```

<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive" version="5">
<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>1e-08</dt>
    <iter>0</iter>
    <subStepping>0</subStepping>
    <subStep>-1</subStep>
    <time>0</time>
    <stopAtIter>0</stopAtIter>
    <isPeriodic>0</isPeriodic>
    <trackEnergy>0</trackEnergy>
    <runInternalConsistencyChecks>1</runInternalConsistencyChecks>
    <selectedBody>-1</selectedBody>
    <flags>0</flags>
    <tags class_id="3" tracking_level="0" version="0">
      <count>5</count>
      <item_version>0</item_version>
      <item>author=~(bchareyre@dt-rv020)</item>
      <item>isoTime=20110211T183253</item>
      <item>id=20110211T183253p6478</item>
      <item>d.id=20110211T183253p6478</item>
      <item>id.d=20110211T183253p6478</item>
    </tags>
    <engines class_id="4" tracking_level="0" version="0">
      <count>1</count>
      <item_version>1</item_version>
      <item class_id="5" tracking_level="0" version="1">
        <px class_id="7" class_name="GravityEngine" tracking_level="1" version="0" object_id="_3">
          <FieldApplier class_id="8" tracking_level="1" version="0" object_id="_3">

```

```
<GlobalEngine class_id="9" tracking_level="1" version="0" object_id="9">
  <Engine class_id="6" tracking_level="1" version="0" object_id="6">
    <Serializable object_id="_6"></Serializable>
    <dead>0</dead>
    <label></label>
  </Engine>
</GlobalEngine>
</FieldApplier>
<gravity class_id="10" tracking_level="0" version="0">
  <x>0</x>
  <y>0</y>
  <z>-9.8100000000000005</z>
</gravity>
</px>
</item>
</engines>
<_nextEngines>
  <count>0</count>
  <item_version>1</item_version>
</_nextEngines>
<bodies class_id="11" tracking_level="0" version="1">
  <px class_id="12" tracking_level="1" version="0" object_id="_7">
    <Serializable object_id="_8"></Serializable>
    <body class_id="13" tracking_level="0" version="0">
      <count>0</count>
      <item_version>1</item_version>
    </body>
  </px>
</bodies>
<interactions class_id="14" tracking_level="0" version="1">
  <px class_id="15" tracking_level="1" version="0" object_id="_9">
    <Serializable object_id="_10"></Serializable>
    <interaction class_id="16" tracking_level="0" version="0">
      <count>0</count>
      <item_version>1</item_version>
    </interaction>
    <serializeSorted>0</serializeSorted>
  </px>
</interactions>
<energy 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>
    <energies class_id="19" tracking_level="0" version="0">
      <size>0</size>
    </energies>
    <names class_id="20" tracking_level="0" version="0">
      <count>0</count>
      <item_version>0</item_version>
    </names>
    <resetStep>
      <count>0</count>
    </resetStep>
  </px>
</energy>
<materials class_id="22" tracking_level="0" version="0">
  <count>0</count>
  <item_version>1</item_version>
</materials>
<bound class_id="23" tracking_level="0" version="1">
  <px class_id="-1"></px>
</bound>
<cell class_id="25" tracking_level="0" version="1">
  <px class_id="26" tracking_level="1" version="0" object_id="_13">
```

```

<Serializable object_id="_14"></Serializable>
<trsf class_id="27" tracking_level="0" version="0">
  <m00>1</m00>
  <m01>0</m01>
  <m02>0</m02>
  <m10>0</m10>
  <m11>1</m11>
  <m12>0</m12>
  <m20>0</m20>
  <m21>0</m21>
  <m22>1</m22>
</trsf>
<refHSize>
  <m00>1</m00>
  <m01>0</m01>
  <m02>0</m02>
  <m10>0</m10>
  <m11>1</m11>
  <m12>0</m12>
  <m20>0</m20>
  <m21>0</m21>
  <m22>1</m22>
</refHSize>
<hSize>
  <m00>1</m00>
  <m01>0</m01>
  <m02>0</m02>
  <m10>0</m10>
  <m11>1</m11>
  <m12>0</m12>
  <m20>0</m20>
  <m21>0</m21>
  <m22>1</m22>
</hSize>
<velGrad>
  <m00>0</m00>
  <m01>0</m01>
  <m02>0</m02>
  <m10>0</m10>
  <m11>0</m11>
  <m12>0</m12>
  <m20>0</m20>
  <m21>0</m21>
  <m22>0</m22>
</velGrad>
<prevVelGrad>
  <m00>0</m00>
  <m01>0</m01>
  <m02>0</m02>
  <m10>0</m10>
  <m11>0</m11>
  <m12>0</m12>
  <m20>0</m20>
  <m21>0</m21>
  <m22>0</m22>
</prevVelGrad>
<homoDeform>3</homoDeform>
</px>
</cell>
<miscParams class_id="28" tracking_level="0" version="0">
  <count>0</count>
  <item_version>1</item_version>
</miscParams>

```

```
<dispParams class_id="29" 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 [69]: s=Sphere()

Yade [70]: s.radius          ## read-access
-> [70]: nan

Yade [71]: s.radius=4.       ## write access

Yade [72]: s.keys()          ## show all available keys
-----
AttributeError                                Traceback (most recent call last)

/home/3S-LAB/bchareyre/yade/yade-test-kdev/yade/doc/sphinx/<ipython console> in <module>()

AttributeError: 'Sphere' object has no attribute 'keys'

Yade [73]: for k in s.keys(): print s[k]  ## iterate over keys, print their values
.....
-----
AttributeError                                Traceback (most recent call last)

/home/3S-LAB/bchareyre/yade/yade-test-kdev/yade/doc/sphinx/<ipython console> in <module>()

AttributeError: 'Sphere' object has no attribute 'keys'

Yade [74]: s.has_key('radius')          ## same as: 'radius' in s.keys()
-----
AttributeError                                Traceback (most recent call last)

/home/3S-LAB/bchareyre/yade/yade-test-kdev/yade/doc/sphinx/<ipython console> in <module>()

AttributeError: 'Sphere' object has no attribute 'has_key'

Yade [75]: s.dict()                  ## show dictionary of both attributes and values
-> [75]: {'color': Vector3(1,1,1), 'highlight': False, 'radius': 4.0, 'wire': False}

## only very rarely needed; calls Serializable::postProcessAttributes(bool deserializing):
Yade [77]: s.postProcessAttributes(False)
-----
AttributeError                                Traceback (most recent call last)

/home/3S-LAB/bchareyre/yade/yade-test-kdev/yade/doc/sphinx/<ipython console> in <module>()
```

```
AttributeError: 'Sphere' object has no attribute 'postProcessAttributes'
```

3.3.5 YADE_CLASS_BASE_DOC_* macro family

There is 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 base macro `YADE_CLASS_BASE_DOC_ATTRS_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_DEPREC_INIT_CTOR_PY(klass,base,doc,attrs,,init,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, but sequences interpreted by c++ compiler must be properly escaped (therefore some backslashes must be doubled, like in $\sigma = \varepsilon E$:

```
":math:``\sigma=\\epsilon E"
```

Use `\n` and `\t` for indentation inside the docstring. Hyperlink the documentation abundantly with `yref` (all references to other classes should be hyperlinks).

See *Sphinx documentation* for syntax details.

attrs Attribute must be written in the form of parenthesized 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

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 viscos.
```

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 Parenthesized 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) { /* constructo
```

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.

Special python constructors

The Python wrapper automatically create 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](#).

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`**
`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 `C`
`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++.

3.3.6 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 currently 4 predefined dispatchers (see [dispatcher-names](#)) and corresponding functor types. They inherit 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_Dem3DofGeom: 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.
    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 `Dem3DofGeom` classes (for different combination of shapes: `Dem3DofGeom_SphereSphere`, `Dem3DofGeom_FacetSphere`, `Dem3DofGeom_WallSphere`), but only provide a `LawFunctor` accepting `Dem3DofGeom`, 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 `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	<code>BoundFunctor</code> , <code>IGeomDispatcher</code>
Material	<code>IPhysDispatcher</code>
IPhys	<code>LawDispatcher</code>
IGeom	<code>LawDispatcher</code>

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!
    /* ... */
};
```

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.

```

    // 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 [78]: Sphere().dispIndex, Facet().dispIndex, Wall().dispIndex
-> [78]: (1, 5, 7)

```

```

Yade [79]: Shape().dispIndex                                # top-level indexable
-> [79]: -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 [80]: Dem3DofGeom().dispHierarchy()                    # parent class of all other Dem3DofGeom_ classes
-> [80]: ['Dem3DofGeom', 'IGeom']

```

```

Yade [81]: Dem3DofGeom_SphereSphere().dispHierarchy(), Dem3DofGeom_FacetSphere().dispHierarchy(), Dem3DofGeom_WallSphere().dispHierarchy()
-> [81]:
(['Dem3DofGeom_SphereSphere', 'Dem3DofGeom', 'IGeom'],
 ['Dem3DofGeom_FacetSphere', 'Dem3DofGeom', 'IGeom'],
 ['Dem3DofGeom_WallSphere', 'Dem3DofGeom', 'IGeom'])

```

```

Yade [82]: Dem3DofGeom_WallSphere().dispHierarchy(names=False) # show numeric indices instead
-> [82]: [6, 3, -1]

```

Dispatchers can also be inspected, using the `.dispMatrix()` method:

```

Yade [83]: ig=IGeomDispatcher([
.....:     Ig2_Sphere_Sphere_Dem3DofGeom(),
.....:     Ig2_Facet_Sphere_Dem3DofGeom(),
.....:     Ig2_Wall_Sphere_Dem3DofGeom()
.....: ])

```

```
Yade [88]: ig.dispMatrix()
-> [88]:
{('Facet', 'Sphere'): 'Ig2_Facet_Sphere_Dem3DofGeom',
 ('Sphere', 'Facet'): 'Ig2_Facet_Sphere_Dem3DofGeom',
 ('Sphere', 'Sphere'): 'Ig2_Sphere_Sphere_Dem3DofGeom',
 ('Sphere', 'Wall'): 'Ig2_Wall_Sphere_Dem3DofGeom',
 ('Wall', 'Sphere'): 'Ig2_Wall_Sphere_Dem3DofGeom'}

Yade [89]: ig.dispMatrix(False)           # don't convert to class names
-> [89]:
{(1, 1): 'Ig2_Sphere_Sphere_Dem3DofGeom',
 (1, 5): 'Ig2_Facet_Sphere_Dem3DofGeom',
 (1, 7): 'Ig2_Wall_Sphere_Dem3DofGeom',
 (5, 1): 'Ig2_Facet_Sphere_Dem3DofGeom',
 (7, 1): 'Ig2_Wall_Sphere_Dem3DofGeom'}
```

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 [90]: ld=LawDispatcher([Law2_Dem3DofGeom_CpmPhys_Cpm()])

Yade [91]: ld.dispMatrix()
-> [91]: {('Dem3DofGeom', 'CpmPhys'): 'Law2_Dem3DofGeom_CpmPhys_Cpm'}

# see how the entry for Dem3DofGeom_SphereSphere will be filled after this request
Yade [93]: ld.dispFunctor(Dem3DofGeom_SphereSphere(),CpmPhys())
-> [93]: <Law2_Dem3DofGeom_CpmPhys_Cpm instance at 0x511d500>

Yade [94]: ld.dispMatrix()
-> [94]:
{('Dem3DofGeom', 'CpmPhys'): 'Law2_Dem3DofGeom_CpmPhys_Cpm',
 ('Dem3DofGeom_SphereSphere', 'CpmPhys'): 'Law2_Dem3DofGeom_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 `G1ShapeFunctor`, `G1BoundFunctor`, `G1IGeomFunctor` and so on. These classes register the type they render using the `RENDERS` macro:

```
class G11_Sphere: public G1ShapeFunctor {
public :
    virtual void go(const shared_ptr<Shape>&,
                    const shared_ptr<State>&,
                    bool wire,
                    const GLViewInfo&
                    );
    RENDERS(Sphere);
    YADE_CLASS_BASE_DOC_STATICATTRS(G11_Sphere,G1ShapeFunctor,"docstring",
    ((Type1,staticAttr1,informativeDefault,"docstring"))
    /* ... */)
};
REGISTER_SERIALIZABLE(G11_Sphere);
```

You can list available functors of a particular type by querying child classes of the base functor:

```
Yade [96]: yade.system.childClasses('G1ShapeFunctor')
-> [96]:
set(['G11_Box',
```

```
'Gl1_ChainedCylinder',
'Gl1_Cylinder',
'Gl1_Facet',
'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.

3.3.7 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-intrusive 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(),En
                             # 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 evaluate 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*.

3.3.8 Logging

Regardless of whether the *optional-libraries* *log4cxx* is used or not, yade provides logging macros.⁶ If *log4cxx* is enabled, these macros internally operate on the local logger instance (named *logger*, but that is hidden for the user); if *log4cxx* is disabled, they send their arguments to standard error output (*cerr*).

Log messages are classified by their *severity*, which is one of *TRACE* (tracing variables), *DEBUG* (generally uninteresting messages useful for debugging), *INFO* (information messages – only use sparingly), *WARN* (warning), *FATAL* (serious error, consider throwing an exception with description instead). Logging level determines which messages will be shown – by default, *INFO* and higher will be shown; if you run yade with *-v* or *-vv*, *DEBUG* and *TRACE* messages will be also enabled (with *log4cxx*).

Every class using logging should create logger using these 2 macros (they expand to nothing if *log4cxx* is not used):

DECLARE_LOGGER; in class declaration body (in the *.hpp* file); this declares static variable *logger*;

CREATE_LOGGER(ClassName); in the implementation file; it creates and initializes that static variable. The logger will be named *yade.ClassName*.

The logging macros are the following:

- *LOG_TRACE*, *LOG_DEBUG*, *LOG_INFO*, *LOG_WARN*, *LOG_ERROR*, *LOG_FATAL* (increasing severity); their argument is fed to the logger stream, hence can contain the *<<* operation:

```
LOG_WARN("Exceeded "<<maxSteps<<" steps in attempts to converge, the result returned will not
```

Every log message is prepended filename, line number and function name; the final message that will appear will look like this:

```
237763 WARN yade.ViscosityIterSolver /tmp/yade/trunk/extra/ViscosityIterSolver.cpp:316 newton
```

The 237763 WARN *yade.ViscosityIterSolver* (microseconds from start, severity, logger name) is added by *log4cxx* and is completely configurable, either programatically, or by using file *~/yade-\$SUFFIX/logging.conf*, which is loaded at startup, if present (FIXME: see more etc user's guide)

- special tracing macros *TRVAR1*, *TRVAR2*, ... *TRVAR6*, which show both variable name and its value (there are several more macros defined inside */lib/base/Logging.hpp*, but they are not generally in use):

```
TRVAR3(var1,var2,var3);  
// will be expanded to:  
LOG_TRACE("var1="<<var1<<" ; var2="<<var2<<" ; var3="<<var3);
```

Note: For performance reasons, optimized builds eliminate *LOG_TRACE* and *LOG_DEBUG* from the code at preprocessor level.

Note: Builds without *log4cxx* (even in debug mode) eliminate *LOG_TRACE* and *LOG_DEBUG*. As there is no way to enable/disable them selectively, the log amount would be huge.

Python provides rudimentary control for the logging system in *yade.log* module (FIXME: ref to docs):

```
Yade [97]: from yade import log
```

```
Yade [98]: log.setLevel('InsertionSortCollider',log.DEBUG) # sets logging level of the yade.InsertionSortCollid
```

```
Yade [99]: log.setLevel('',log.WARN) # sets logging level of all yade.* loggers (they in
```

⁶ Because of (seemingly?) no upstream development of *log4cxx* and a few problems it has, Yade will very likely move to the hypothetical *boost::logging* library once it exists. The logging code will not have to be changed, however, as the *log4cxx* logic is hidden behind these macros.

As of now, there is no python interface for performing logging into log4cxx loggers themselves.

3.3.9 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 [100]: TriaxialTest(numberOfGrains=100).load()

Yade [101]: O.engines[0].label='firstEngine'    ## labeled engines will show by labels in the stats table

Yade [102]: import yade.timing;
Yade [103]: O.timingEnabled=True

Yade [104]: yade.timing.reset()                  ## not necessary if used for the first time

Yade [105]: O.run(50); O.wait()

Yade [106]: yade.timing.stats()
```

Name	Count	Time	Rel. time
"firstEngine"	50	29us	0.39%
InsertionSortCollider	50	2655us	35.52%
InteractionLoop	50	3054us	40.86%
GlobalStiffnessTimeStepper	2	15us	0.20%
TriaxialCompressionEngine	50	353us	4.73%
TriaxialStateRecorder	3	158us	2.12%
NewtonIntegrator	50	1209us	16.17%
TOTAL		7476us	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. It is made for timing loops (functors' loop is in their respective dispatcher) and stores cummulative 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. `Engine::timingDeltas` must point to an instance of `TimingDeltas` (preferably instantiate `TimingDeltas` in the constructor):

```

// header file
class Law2_Dem3DofGeom_CpmPhys_Cpm: public LawFunctor {
    /* ... */
    YADE_CLASS_BASE_DOC_ATTRS_CTOR(Law2_Dem3DofGeom_CpmPhys_Cpm, LawFunctor, "docstring",
        /* attrs */,
        /* constructor */
        timingDeltas=shared_ptr<TimingDeltas>(new TimingDeltas);
    );
    // ...
};

```

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_Dem3DofGeom_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");
}

```

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	9449µs	0.01%
BoundDispatcher	400	1171770µs	1.15%
InsertionSortCollider	400	9433093µs	9.24%
IGeomDispatcher 400	15177607µs	14.87%	
IPhysDispatcher 400	9518738µs	9.33%	
LawDispatcher	400	64810867µs	63.49%
Law2_Dem3DofGeom_CpmPhys_Cpm			
setup	4926145	7649131µs	15.25%
geom	4926145	23216292µs	46.28%
material	4926145	8595686µs	17.14%
rest	4926145	10700007µs	21.33%
TOTAL		50161117µs	100.00%
NewtonIntegrator	400	1866816µs	1.83%
"strainer"	400	21589µs	0.02%
"plotDataCollector"	160	64284µs	0.06%
"damageChecker"	9	3272µs	0.00%
TOTAL		102077490µs	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 disproportionately longer time than the rest of engine; in the output presented above, `LawDispatcher` takes almost 1/3 of total simulation time in average, but the number would be twice or thrice lower typically (note that each checkpoint was timed almost 5 million times in this particular case).

3.3.10 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*.

`G11_*` 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.

3.4 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.

3.4.1 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 internally 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.

3.4.2 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 ($0 \dots n$ indexability),
- store `shared_ptr`, not objects themselves,
- *no* mutual exclusion for insertion/removal (this must be assured by the caller, 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 **FOREACH** macro (shorthand for **BOOST_FOREACH**):

```
FOREACH(const shared_ptr<Body>& b, *scene->bodies){  
    if(!b) continue; // skip deleted bodies  
    /* do something here */  
}
```

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 are silently skipped:

```
Yade [108]: O.bodies.append([Body(),Body(),Body()])
-> [108]: [0, 1, 2]
```

```
Yade [109]: O.bodies.erase(1)
-> [109]: True
```

```
Yade [110]: [b.id for b in O.bodies]
-> [110]: [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;
    /* ... */
}
```

3.4.3 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 elements 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
```

```
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 `FOREACH` macro:

```
FOREACH(const shared_ptr<Interaction>& i, *scene->interactions){
    if(!i->isReal()) continue;
    /* ... */
}
```

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:

```
inr 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;
    /* ... */
}
```

3.4.4 ForceContainer

`ForceContainer` holds “generalized forces”, i.e. forces, torques, (explicit) displacements 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 write 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 cumulative 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 Vector3& f=scene->forces.getForce(id);
```

Synchronization is handled automatically if values are read from python:

```
Yade [111]: O.bodies.append(Body())
-> [111]: 1
```

```
Yade [112]: O.forces.addF(0,Vector3(1,2,3))
```

```
Yade [113]: O.forces.f(0)
-> [113]: Vector3(1,2,3)
```

```
Yade [114]: O.forces.f(100)
```

```
-----
IndexError                                Traceback (most recent call last)

/home/3S-LAB/bchareyre/yade/yade-test-kdev/yade/doc/sphinx/<ipython console> in <module>()

IndexError: Body id out of range.
```

3.4.5 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 still remain 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` can decide that an interaction should be removed (such as if bodies get too far apart for non-cohesive laws; or in case of complete damage for damage models). This is done by calling

```
InteractionContainer::requestErase(id1,id2)
```

Such interaction will not be deleted immediately, but will be reset to potential state. At next step, the collider will call `InteractionContainer::erasePending`, which will only completely erase interactions the collider indicates; 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.

3.4.6 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 to construct `Body` objects from functions such as `utils.sphere` only by knowing the required `Material` type, enforcing the expectation of the model implementor.

3.5 Runtime structure

3.5.1 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 logging, 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.

3.5.2 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 accessible 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:

SerializableSingleton Handles serialization/deserialization; it is not used anywhere except for the serialization code proper.

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.

Omega

The **Omega** class handles all simulation-related functionality: loading/saving, running, pausing.

In python, the wrapper class to the singleton is instantiated ⁷ 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`.

To give an overview of (some) variables:

Python	c++
<code>Omega.iter</code>	<code>Scene::iter</code>
<code>Omega.dt</code>	<code>Scene::dt</code>
<code>Omega.time</code>	<code>Scene::time</code>
<code>Omega.realtime</code>	<code>Omega::getRealTime()</code>
<code>Omega.stopAtIter</code>	<code>Scene::stopAtIter</code>

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.

3.5.3 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`:

⁷ It is understood that instantiating `Omega()` in python only instantiates the wrapper class, not the singleton itself.

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()`.

3.6 Python framework

3.6.1 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 [115]: f1=ForceEngine()
```

```
Yade [116]: f1.subscribedBodies=[0,4,5]
```

```
Yade [117]: f1.force=Vector3(0,-1,-2)
```

and

```
Yade [118]: f2=ForceEngine(ids=[0,4,5],force=Vector3(0,-1,-2))
```

```
Yade [119]: print f1.dict()
{'force': Vector3(0,-1,-2), 'ids': [], 'dead': False, 'label': ''}
```

```
Yade [120]: print f2.dict()
{'force': Vector3(0,-1,-2), 'ids': [0, 4, 5], 'dead': False, '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).

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 [121]: s=Sphere()
```

```
Yade [122]: s.name, s.__class__.__name__
```

```
-----
AttributeError
```

```
Traceback (most recent call last)
```

```
/home/3S-LAB/bchareyre/yade/yade-test-kdev/yade/doc/sphinx/<ipython console> in <module>()
```

```
AttributeError: 'Sphere' object has no attribute 'name'
```

3.6.2 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: `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.

3.6.3 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.

3.6.4 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;⁸

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

⁸ 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 by hand, to be found in `py/mathWrap/miniEigen.cpp`; this module is imported at startup.

“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 slightly more complicated `add_property` syntax, as explained at the page referenced.

3.7 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.

3.7.1 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 occurrences, moved 0 files and 0 directories
Update python scripts (if wanted) by running: perl -pi -e 's/\bOldClassName\b/NewClassName/g' `ls **/*.py |grep
```

This has the following effects:

1. If file or directory has basename `OldClassName` (plus extension), it will be renamed using `bzr`.
2. All occurrences 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 disrupting your script’s functioning:

```
Yade [3]: SimpleViscoelasticMat()
/usr/local/lib/yade-trunk/py/yade/__init__.py:1: DeprecationWarning: Class `SimpleViscoelasticMat' was :
-> [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.

3.7.2 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.

3.8 Debian packaging instructions

In order to make parallel installation of several Yade version possible, we adopted similar strategy as e.g. gcc packagers in Debian did:

1. Real Yade packages are named `yade-0.30` (for stable versions) or `yade-bzr2341` (for snapshots).
2. They provide `yade` or `yade-snapshot` virtual packages respectively.
3. Each source package creates several installable packages (using `bzr2341` as example version):
 - (a) `yade-bzr2341` with the optimized binaries; the executable binary is `yade-bzr2341` (`yade-bzr2341-multi`, ...)
 - (b) `yade-bzr2341-dbg` with debug binaries (debugging symbols, non-optimized, and with crash handlers); the executable binary is `yade-bzr2341-dbg`
 - (c) `yade-bzr2341-doc` with sample scripts and some documentation (see [bug #398176](#) however)
 - (d) (future?) `yade-bzr2341-reference` with reference documentation (see [bug #401004](#))
4. Using [Debian alternatives](#), the highest installed package provides additionally commands without the version specification like `yade`, `yade-multi`, ... as aliases to that version's binaries. (`yade-dbg`, ... for the debuggin packages). The exact rule is:
 - (a) Stable releases have always higher priority than snapshots
 - (b) Higher versions/revisions have higher priority than lower versions/revisions.

3.8.1 Prepare source package

Debian packaging files are located in `debian/` directory. They contain build recipe `debian/rules`, dependency and package declarations `debian/control` and maintainer scripts. Some of those files are only provided as templates, where some variables (such as version number) are replaced by special script.

The script `scripts/debian-prep` processes templates in `debian/` and creates files which can be used by debian packaging system. Before running this script:

1. If you are releasing stable version, make sure there is file named `RELEASE` containing single line with version number (such as `0.30`). This will make `scripts/debian-prep` create release packages. In absence of this file, snapshots packaging will be created instead. Release or revision number (as detected by running `bzr revno` in the source tree) is stored in `VERSION` file, where it is picked up during package build and embedded in the binary.
2. Find out for which debian/ubuntu series your package will be built. This is the name that will appear on the top of (newly created) `debian/changelog` file. This name will be usually `unstable`, `testing` or `stable` for debian and `karmic`, `lucid` etc for ubuntu. When package is uploaded to Launchpad's build service, the package will be built for this specified release.

Then run the script from the top-level directory, giving series name as its first (only) argument:

```
$ scripts/debian-prep lucid
```

After this, signed debian source package can be created:

```
$ debuild -S -sa -k62A21250 -I -Iattic
```

(`-k` gives GPG key identifier, `-I` skips `.bzr` and similar directories, `-Iattic` will skip the useless `attic` directory).

3.8.2 Create binary package

Local in-tree build Once files in `debian/` are prepared, packages can be build by issuing:: `$ fakeroot debian/rules binary`

Clean system build Using `pbuilder` system, package can be built in a chroot containing clean debian/ubuntu system, as if freshly installed. Package dependencies are automatically installed and package build attempted. This is a good way of testing packaging before having the package built remotely at Launchpad. Details are provided at [wiki page](#).

Launchpad build service Launchpad provides service to compile package for different ubuntu releases (series), for all supported architectures, and host archive of those packages for download via APT. Having appropriate permissions at Launchpad (verified GPG key), source package can be uploaded to yade's archive by:

```
$ dput ppa:yade-users/ppa ../yade-bzr2341_1_source.changes
```

After several hours (depending on load of Launchpad build farm), new binary packages will be published at <https://launchpad.net/~yade-users/+archive/ppa>.

This process is well documented at <https://help.launchpad.net/Packaging/PPA>.

Chapter 4

Installation

Yade can be installed from packages (precompiled 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 source code.

4.1 Packages

Packages are (as of now) provided for several Ubuntu versions from [Yade package archive](#). Different version of Yade can be installed alongside each other. The **yade** virtual package always depends on the latest stable package, while **yade-snapshot** will pull the latest snapshot package. To install quickly, run the following:

```
sudo add-apt-repository ppa:yade-users/ppa
sudo add-apt-repository ppa:yade-users/external # optional (updates of other packages)
sudo apt-get update
sudo apt-get install yade
```

More detailed instructions are available at the [archive page](#)

4.2 Source code

4.2.1 Download

If you want to install from source, you can install 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 version will not be modified (except for updates due to critical and easy-to-fix bugs), but they are in a more stabilized state than trunk generally.

1. Releases can be downloaded from the [download page](#), as compressed archive. Uncompressing the archive gives you a directory with the sources.
2. development version (trunk) can be obtained from the [code repository](#) at Launchpad. We use **Bazaar** (the **bzr** command) for code management (install the **bzr** package in your distribution):

```
bzr checkout lp:yade
```

will download the whole code repository of **trunk**. Check out [Quick Bazaar tutorial](#) wiki page for more. For those behind firewall, [daily snapshot](#) of the repository (as compressed archive) is provided.

Release and trunk sources are compiled in the same way.

4.2.2 Prerequisites

Yade relies on a number of external software to run; its installation is checked before the compilation starts.

- `scons` build system
- `gcc` compiler (g++); other compilers will not work; you need `g++>=4.2` for openMP support
- `boost` 1.35 or later
- `qt3` library
- `freeglut3`
- `libQGLViewer`
- `python`, `numpy`, `ipython`
- `matplotlib`
- `eigen2` algebra library
- `gdb` debugger
- `sqlite3` database engine
- `Loki` library
- `VTK` library (optional but recommended)

Most of the list above is very likely already packaged for your distribution. In Ubuntu, it can be all installed by the following command (cut&paste to the terminal):

```
sudo apt-get install scons freeglut3-dev libloki-dev \  
libboost-date-time-dev libboost-filesystem-dev libboost-thread-dev \  
libboost-regex-dev fakeroot dpkg-dev build-essential g++ \  
libboost-iostreams-dev liblog4cxx10-dev python-dev libboost-python-dev ipython \  
python-matplotlib libsqlite3-dev python-numeric python-tk gnuplot doxygen \  
libgts-dev python-pygraphviz libvtk5-dev python-scientific bzr libeigen2-dev \  
binutils-gold python-xlib python-qt4 pyqt4-dev-tools \  
libqglviewer-qt4-dev python-imaging libjs-jquery
```

command line (cut&paste to the terminal under root privileges) for Fedora (not good tested yet!):

```
yum install scons qt3-devel freeglut-devel boost-devel boost-date-time \  
boost-filesystem boost-thread boost-regex fakeroot gcc gcc-c++ boost-iostreams \  
log4cxx log4cxx-devel python-devel boost-python ipython python-matplotlib \  
sqlite-devel python-numeric ScientificPython-tk gnuplot doxygen gts-devel \  
graphviz-python vtk-devel ScientificPython bzr eigen2-devel libQGLViewer-devel \  
loki-lib-devel python-xlib PyQt4 PyQt4-devel python-imaging
```

4.2.3 Compilation

Inside the directory where you downloaded the sources (ex “yade” if you use bazaar), install Yade to your home directory (without root privileges):

```
scons PREFIX=/home/username/YADE
```

If you have a machine that you are the only user on, you can instead change permission on `/usr/local` and install subsequently without specifying the `PREFIX`:

```
sudo chown user: /usr/local    # replace "user" with your login name  
scons
```

There is a number of options for compilation you can change; run `scons -h` to see them (see also *scons-parameters* in the *Programmer's manual*)

The compilation process can take a long time, be patient.

Decreasing RAM usage during compilation

Yade demands a large amount of memory for compilation (due to extensive template use). If you have less than 2GB of RAM, it will be, you might encounter difficulties such as the computer being apparently stalled, compilation taking very long time (hours) or erroring out. This command will minimize RAM usage, but the compilation will take longer – only one file will be compiled simultaneously and files will be “chunked” together one by one:

```
scons jobs=1 chunkSize=1
```


Bibliography

- [Allen1989] M. P. Allen, D. J. Tildesley (1989), **Computer simulation of liquids**. Clarendon Press.
- [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.