Yade Documentation

Václav Šmilauer, Emanuele Catalano, Bruno Chareyre, Sergei Dorofeenko, Jerome Duriez, Anton Gladky, Janek Kozicki, Chiara Modenese, Luc Scholtès, Luc Sibille, Jan Stránský, Klaus Thoeni

Release 0.80.1, July 5, 2012

Authors

Václav Šmilauer
University of Innsbruck

Emanuele Catalano
Grenoble INP, UJF, CNRS, lab. 3SR

Bruno Chareyre
Grenoble INP, UJF, CNRS, lab. 3SR

Sergei Dorofeenko
IPCP RAS, Chernogolovka

Jerome Duriez
Grenoble INP, UJF, CNRS, lab. 3SR

Anton Gladky
TU Bergakademie Freiberg

Janek Kozicki
Gdansk University of Technology - lab. 3SR Grenoble University

Chiara Modenese
University of Oxford

Luc Scholtès
Grenoble INP, UJF, CNRS, lab. 3SR

Luc Sibille
University of Nantes, lab. GeM

Jan Stránský
CVUT Prague

Klaus Thoeni The University of Newcastle (Australia)

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.

Contents

Note: Please consult changes to Yade documention with documentation manager (whoever that is), even if you have commit permissions.

See older tentative contents

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 topis, 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 command [2]):

```
$ yade
Welcome to Yade bzr1984
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 (`bzr1984`); 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 hello.py
Welcome to Yade bzr1986
TCP python prompt on localhost:9001, auth cookie `askcsu'
TCP info provider on localhost:21000
Running script hello.py                                           ## the script is being run
Hello world!                                                      ## output from the script
[[ ^L clears screen, ^U kills line. F12 controller, F11 3d view, F10 both, F9 generator, F8 plot. ]]
Yade [1]:
```

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

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

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

Options:

--version	show program's version number and exit
-h, --help	show this help message and exit
-j THREADS, --threads=THREADS	Number of OpenMP threads to run; defaults to 1. Equivalent to setting OMP_-NUM_THREADS environment variable.
--cores=CORES	Set number of OpenMP threads (as –threads) and in addition set affinity of threads to the cores given.
--update	Update deprecated class names in given script(s) using text search & replace. Changed files will be backed up with ~ suffix. Exit when done without running any simulation.
--nice=NICE	Increase nice level (i.e. decrease priority) by given number.
-x	Exit when the script finishes
-n	Run without graphical interface (equivalent to unsetting the DISPLAY environment variable)
--generate-manpage=FILE	Generate man page documenting this program and exit
--rebuild	Re-run build in the source directory, then run the updated yade with the same command line except –rebuild. The build profile for this build (deb) and its stored parameters will be used.
--test	Run regression test suite and exit; the exists status is 0 if all tests pass, 1 if a test fails and 2 for an unspecified exception.

[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.

--debug	Run the debug build, if available.
--checks	Run a series of user-defined check tests as described in /build/buildd/yade-daily-1+3021+27~lucid1/scripts/test/checks/README
--performance	Starts a test to measure the productivity
--no-gdb	Do not show backtrace when yade crashes (only effective with –debug).

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:

```
In [1]: TriaxialTest(numberOfGrains=200).load()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/build/buildd/yade-0.80.1/debian/inst/lib/yade/py/yade/__init__.pyc in <module>()

NameError: name 'yade' is not defined

In [2]: len(O.bodies)
 Out[2]: 0
```

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`:

```
In [1]: O.iter
 Out[1]: 0

In [2]: O.time
 Out[2]: 0.0

In [3]: O.dt=1e-4

In [4]: O.step()

In [5]: O.iter
 Out[5]: 1

In [6]: O.time
 Out[6]: 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.

```
In [1]: O.run()

In [2]: O.pause()

In [3]: O.iter
 Out[3]: 488

In [4]: O.run(100000,True)

In [5]: O.iter
 Out[5]: 100488

In [6]: O.stopAtIter=500000

In [7]: O.wait()

In [8]: O.iter
 Out[8]: 100488
```

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.

```
In [1]: O.save('/tmp/a.xml.bz2')

In [2]: O.reload()

In [4]: 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:

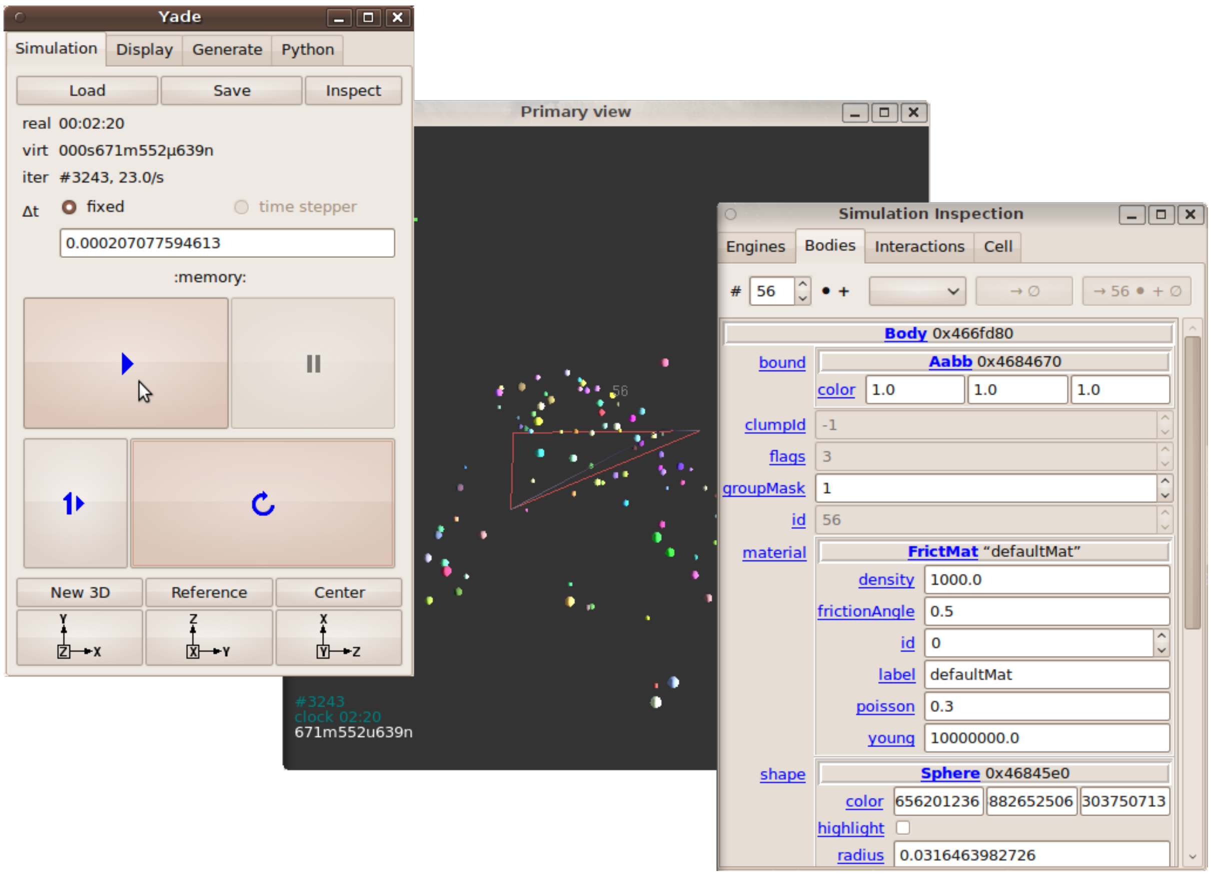
```
In [1]: O.saveTmp()

In [2]: O.loadTmp()

In [3]: O.saveTmp('init') ## named memory slot

In [4]: 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.



real 00:02:20
virt 000s671m552μ639n
iter #3243, 23.0/s
Δt [✓ fixed] time stepper
0.000207077594613

#3243
clock 02:20
671m552u639n

# 56

Body 0x466fd80
bound: Aabb 0x4684670
color 1.0 1.0 1.0
clumpId -1
flags 3
groupMask 1
id 56
material: FrictMat "defaultMat"
density 1000.0
frictionAngle 0.5
id 0
label defaultMat
poisson 0.3
young 10000000.0
shape: Sphere 0x46845e0
color 656201236 882652506 303750713
highlight
radius 0.0316463982726

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:

Data components

Bodies

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

Each Body comprises the following:

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

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

State contains state 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:

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

In [2]: s.shape, s.state, s.mat, s.bound
 Out[2]:
(<Sphere instance at 0xa9a18f8>,
 <State instance at 0xafb8eb8>,
 <FrictMat instance at 0xd42c220>,
 None)

In [3]: s.state.pos
 Out[3]: Vector3(0,0,0)

In [4]: s.shape.radius
 Out[4]: 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.

```
In [1]: O.bodies.append(s)
 Out[1]: 0

In [2]: O.bodies.append(utils.sphere([0,0,2],.5))
 Out[2]: 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:

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

In [2]: O.bodies[100]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/build/buildd/yade-0.80.1/debian/inst/lib/yade/py/yade/__init__.pyc in <module>()
----> 1 O.bodies[100]

IndexError: Body id out of range.
```

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

```
In [1]: O.bodies.append(s)
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/build/buildd/yade-0.80.1/debian/inst/lib/yade/py/yade/__init__.pyc in <module>()
----> 1 O.bodies.append(s)

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:

```
In [1]: 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:

```
In [1]: O.interactions[0,1]
 Out[1]: <Interaction instance at 0xbfb3450>

In [2]: O.interactions[1,0]     # order of ids is not important
 Out[2]: <Interaction instance at 0xbfb3450>

In [3]: i=O.interactions[0,1]

In [4]: i.id1,i.id2
```

```
 Out[4]: (0, 1)

In [5]: i.geom
 Out[5]: <Dem3DofGeom_SphereSphere instance at 0xb4fc838>

In [6]: i.phys
 Out[6]: <FrictPhys instance at 0xad725b0>

In [7]: O.interactions[100,10111]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/build/buildd/yade-0.80.1/debian/inst/lib/yade/py/yade/__init__.pyc in <module>()
----> 1 O.interactions[100,10111]

IndexError: No such interaction
```

Generalized forces

Generalized forces include force, torque and forced displacement and rotation; they are stored only temporarliy, 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.

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

Function components

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

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

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

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
```

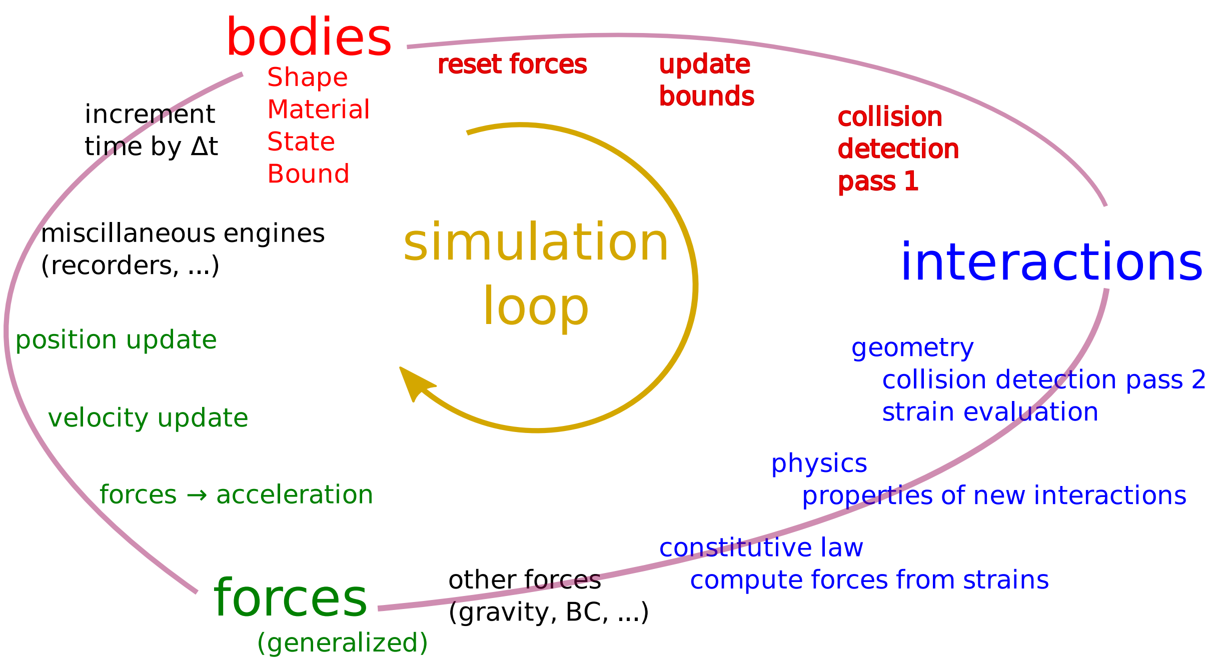


Figure 1.1: Typical simulation loop; each step begins at body-cented bit at 11 o'clock, continues with interaction bit, force application bit, miscillanea and ends with time update.

```
        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 obejct is treated by the dispatcher.

Dispatchers and functors

For approximate collision detection (pass 1), we want to compute bounds for all bodies in the simulation; suppose we want bound of type axis-aligned bounding box. Since the exact algorithm is different depending on particular shape, we need to provide functors for handling all specific cases. 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 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.

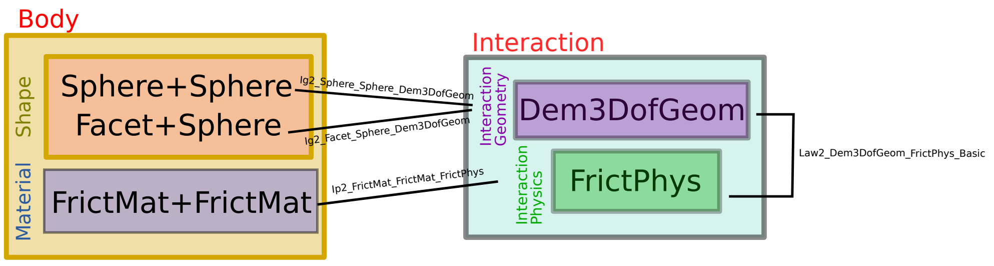


Figure 1.2: Chain of functors producing and accepting certain types. In the case shown, the `Ig2` functors produce Dem3DofGeom instances from all handled Shape combinations; the `Ig2` functor produces FrictMat. The constitutive law functor `Law2` accepts the combination of types produced. Note that the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

Chapter 2

Tutorial

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

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

A few minor bugs were discovered during the course. They were all fixed in rev. 2640 of Yade which is therefore the minimum recommended version to run the examples (notably, 0.60 will not work).

2.1 Introduction

Slides Yade: past, present and future (updated version)

2.2 Hands-on

2.2.1 Shell basics

Directory tree

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

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

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

Shell navigation

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

Moving around

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

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

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

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

Keys

Useful keys on the command-line are:

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

Running programs

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

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

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

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

Starting yade

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

```
user@machine:~$ yade
Welcome to Yade bzr2616
TCP python prompt on localhost:9002, auth cookie `adcusk'
XMLRPC info provider on http://localhost:21002
[[ ^L clears screen, ^U kills line. F12 controller, F11 3d view, F10 both, F9 generator, F8 plot. ]]
Yade [1]:                                          #### hit ^D to exit
Do you really want to exit ([y]/n)?
Yade: normal exit.
```

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

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

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

For a complete help, see `man yade`

Exercises

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

2.2.2 Python basics

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

Numerical operations and modules:

Variables:

Sequences

Lists

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

Lists can be created in various ways:

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

Tuples

Tuples are constant sequences:

Dictionaries

Mapping from keys to values:

Functions, conditionals

Exercises

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

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

2.2.3 Yade basics

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

Particles

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

Singles

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

1. Create Body
2. Set Body.shape to be an instance of Sphere with some given radius
3. Set Body.material (last-defined material is used, otherwise a default material is created)
4. Set position and orientation in Body.state, compute mass and moment of inertia based on Material and Shape

In order to avoid such tasks, shorthand functions are defined in the utils module; to mention a few of them, they are utils.sphere, utils.facet, utils.wall.

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

A particle object is not yet part of the simulation; in order to do so, a special function is called:

Packs

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

Boundaries

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

There are other conveinence functions (like utils.facetBox for creating closed or open rectangular box, or family of ymport functions)

Look inside

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

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

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

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

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

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

Exercises

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

Engines

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

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

```
O.engines=[                  # newlines and indentations are not important until the brace is closed
        ForceResetter(),
        InsertionSortCollider([Bo1_Sphere_Aabb(),Bo1_Wall_Aabb()]),
        InteractionLoop(           # dtto for the parenthesis here
                [Ig2_Sphere_Sphere_L3Geom_Inc(),Ig2_Wall_Sphere_L3Geom_Inc()],
                [Ip2_FrictMat_FrictMat_FrictPhys()],
                [Law2_L3Geom_FrictPhys_ElPerfPl()]
        ),
        GravityEngine(gravity=(0,0,-9.81)),              # 9.81 is the gravity acceleration, and we say that
        NewtonIntegrator(damping=.2,label='newton')      # define a name under which we can access this engine
]
```

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

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

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

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

```
O.saveTmp()
```

Exercises

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

See Also:

The *Bouncing sphere* example shows a basic simulation.

2.3 Data mining

2.3.1 Read

Local data

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

1. O.engines

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

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

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

2. O.bodies

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

   ```
   O.bodies[0]                                                # first body
   [b.shape.radius in O.bodies if isinstance(b.shape,Sphere)] # list of radii of all spherical bodies
   sum([b.state.mass for b in O.bodies])                      # sum of masses of all bodies
   ```

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

3. O.force

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

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

4. O.interactions

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

   ```
   O.interactions[0,1]   # interactions of #0 with #1
   O.interactions[1,0]   # the same object
   O.bodies[0].intrs     # all interactions of body #0
   ```

Labels

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

Exercises

1. Find meaning of this expression:

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

2. Run the gravity deposition script, pause after a few seconds of simulation. Write expressions that compute
   (a) kinetic energy $\sum \frac{1}{2} m_i |v_i|^2$
   (b) average mass (hint: use numpy.average)
   (c) maximum z-coordinate of all particles
   (d) number of interactions of body #1

Global data

Useful measures of what happens in the simulation globally:

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

porosity ratio of void volume and total volume; computed with utils.porosity.

coordination number average number of interactions per particle, utils.avgNumInteractions

stress tensor (periodic boundary conditions) averaged force in interactions, computed with utils.normalShearStressTensor and utils.stressTensorOfPeriodicCell

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

Energies

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

```
O.trackEnergy=True
```

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

2.3.2 Save

PyRunner

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

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

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

```
def myFunction():
        '''Print step number, and pause the simulation is unbalanced force is smaller than 0.05.'''
        print O.iter
        if utils.unbalancedForce()<0.05:
                print 'Unbalanced force is smaller than 0.05, pausing.'
                O.pause()
O.engines=[
        # ...
        PyRunner(command='myFunction()',iterPeriod=100) # call myFunction every 100 steps
]
```

Exercises

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

Keeping history

Yade provides the plot module used for storing and plotting variables (plotting itself will be discussed later). Periodic storing of data is done with PyRunner and the plot.addData function, for instance:

```
from yade import plot
O.engines=[  # ...,
        PyRunner(command='addPlotData()',realPeriod=2)                # call the addPlotData function every 2
]
def addPlotData():
        # this function adds current values to the history of data, under the names specified
        plot.addData(i=O.iter,t=O.time,Ek=utils.kineticEnergy(),coordNum=utils.avgNumInteractions(),unForce=uti
```

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

2.3.3 Plot

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

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

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

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

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

Energy plots

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

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

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

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

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

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

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

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

Exercises

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

See Also:

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

2.4 Towards geomechanics

See Also:

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

2.4.1 Parametric studies

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

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

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

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

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

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

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

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

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

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

```
        import sys
        sys.exit(0)                                     # exit the program
```

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

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

The *parameter table* is a simple text-file, where each line specifies a simulation to run:

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

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

```
user@machine:~$ yade-batch parameters.table simulation.py
```

Exercises

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

2.4.2 Boundary

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

Supports

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

Periodic

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

The periodic cell is described by its reference size of box aligned with global axes, and current transformation, which can capture stretch, shear and rotation. Deformation is prescribed via velocity gradient, which updates the transformation before the next step. Homothetic deformation can smear velocity gradient accross the cell, making the boundary dissolve in the whole cell.

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

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

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

```
from yade import pack,plot
sp=pack.SpherePack()
rMean=.05
sp.makeCloud((0,0,0),(1,1,1),rMean=rMean,periodic=True)
sp.toSimulation()
O.engines=[
   ForceResetter(),
   InsertionSortCollider([Bo1_Sphere_Aabb()],verletDist=.05*rMean),
   InteractionLoop([Ig2_Sphere_Sphere_L3Geom()],[Ip2_FrictMat_FrictMat_FrictPhys()],[Law2_L3Geom_FrictPhys_ElPe
   NewtonIntegrator(damping=.6),
   PeriTriaxController(goal=(-1e6,-1e6,-.1),stressMask=0b011,maxUnbalanced=.2,doneHook='goalReached()',label='t
   PyRunner(iterPeriod=100,command='addPlotData()')
]
O.dt=.5*utils.PWaveTimeStep()
O.trackEnergy=True
def goalReached():
   print 'Goal reached, strain',triax.strain,' stress',triax.stress
   O.pause()
def addPlotData():
   plot.addData(sx=triax.stress[0],sy=triax.stress[1],sz=triax.stress[2],ex=triax.strain[0],ey=triax.strain[1],
      i=O.iter,unbalanced=utils.unbalancedForce(),
      totalEnergy=O.energy.total(),**O.energy       # plot all energies
   )
plot.plots={'i':(('unbalanced','go'),None,'kinetic'),' i':('ex','ey','ez',None,'sx','sy','sz'),'i ':(O.energy.k
plot.plot()
O.saveTmp()
O.run()
```

2.5 Advanced & more

2.5.1 Particle size distribution

See *Periodic triaxial test*

2.5.2 Clumps

Clump; see *Periodic triaxial test*

2.5.3 Testing laws

LawTester, scripts/test/law-test.py

2.5.4 New law

2.5.5 Visualization

See the example *3d postprocessing*

- VTKRecorder & Paraview
- qt.SnapshotEngine

2.6 Examples

2.6.1 Bouncing sphere

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

# DATA COMPONENTS

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

# FUNCTIONAL COMPONENTS

# simulation loop -- see presentation for the explanation
O.engines=[
   ForceResetter(),
   InsertionSortCollider([Bo1_Sphere_Aabb()]),
   InteractionLoop(
      [Ig2_Sphere_Sphere_L3Geom()],        # collision geometry
      [Ip2_FrictMat_FrictMat_FrictPhys()], # collision "physics"
      [Law2_L3Geom_FrictPhys_ElPerfPl()]   # contact law -- apply forces
   ),
   # apply gravity force to particles
   GravityEngine(gravity=(0,0,-9.81)),
   # damping: numerical dissipation of energy
   NewtonIntegrator(damping=0.1)
]

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

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

2.6.2 Gravity deposition

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

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

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

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

```
# add the sphere pack to the simulation
sp.toSimulation()

O.engines=[
   ForceResetter(),
   InsertionSortCollider([Bo1_Sphere_Aabb(),Bo1_Facet_Aabb()]),
   InteractionLoop(
      # handle sphere+sphere and facet+sphere collisions
      [Ig2_Sphere_Sphere_L3Geom(),Ig2_Facet_Sphere_L3Geom()],
      [Ip2_FrictMat_FrictMat_FrictPhys()],
      [Law2_L3Geom_FrictPhys_ElPerfPl()]
   ),
   GravityEngine(gravity=(0,0,-9.81)),
   NewtonIntegrator(damping=0.4),
   # call the checkUnbalanced function (defined below) every 2 seconds
   PyRunner(command='checkUnbalanced()',realPeriod=2),
   # call the addPlotData function every 200 steps
   PyRunner(command='addPlotData()',iterPeriod=100)
]
O.dt=.5*utils.PWaveTimeStep()

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

# if the unbalanced forces goes below .05, the packing
# is considered stabilized, therefore we stop collected
# data history and stop
def checkUnbalanced():
   if utils.unbalancedForce()<.05:
      O.pause()
      plot.saveDataTxt('bbb.txt.bz2')
      # plot.saveGnuplot('bbb') is also possible

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

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

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

O.saveTmp()
```

2.6.3 Oedometric test

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

# The components of the batch are:
# 1. table with parameters, one set of parameters per line (ccc.table)
# 2. utils.readParamsFromTable which reads respective line from the parameter file
# 3. the simulation muse be run using yade-batch, not yade
```

```
#
# $ yade-batch --job-threads=1 03-oedometric-test.table 03-oedometric-test.py
#

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

# create box with free top, and ceate loose packing inside the box
from yade import pack, plot
O.bodies.append(utils.geom.facetBox((.5,.5,.5),(.5,.5,.5),wallMask=31))
sp=pack.SpherePack()
sp.makeCloud((0,0,0),(1,1,1),rMean=rMean,rRelFuzz=rRelFuzz)
sp.toSimulation()

O.engines=[
   ForceResetter(),
   # sphere, facet, wall
   InsertionSortCollider([Bo1_Sphere_Aabb(),Bo1_Facet_Aabb(),Bo1_Wall_Aabb()]),
   InteractionLoop(
      # the loading plate is a wall, we need to handle sphere+sphere, sphere+facet, sphere+wall
      [Ig2_Sphere_Sphere_L3Geom(),Ig2_Facet_Sphere_L3Geom(),Ig2_Wall_Sphere_L3Geom()],
      [Ip2_FrictMat_FrictMat_FrictPhys()],
      [Law2_L3Geom_FrictPhys_ElPerfPl()]
   ),
   GravityEngine(gravity=(0,0,-9.81)),
   NewtonIntegrator(damping=0.5),
   # the label creates an automatic variable referring to this engine
   # we use it below to change its attributes from the functions called
   PyRunner(command='checkUnbalanced()',realPeriod=2,label='checker'),
]
O.dt=.5*utils.PWaveTimeStep()

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

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

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

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

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

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

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

Batch table

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

2.6.4 Periodic simple shear

```
# encoding: utf-8

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


# setup the periodic boundary
O.periodic=True
O.cell.refSize=(2,2,2)

from yade import pack,plot

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

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

# simulation loop (will be run at every step)
O.engines=[
   ForceResetter(),
   InsertionSortCollider([Bo1_Sphere_Aabb()]),
   InteractionLoop(
      [Ig2_Sphere_Sphere_L3Geom()],
      [Ip2_FrictMat_FrictMat_FrictPhys()],
      [Law2_L3Geom_FrictPhys_ElPerfPl()]
   ),
   NewtonIntegrator(damping=.4),
   # run checkStress function (defined below) every second
   # the label is arbitrary, and is used later to refer to this engine
   PyRunner(command='checkStress()',realPeriod=1,label='checker'),
   # record data for plotting every 100 steps; addData function is defined below
   PyRunner(command='addData()',iterPeriod=100)
]

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

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

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

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

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

# called periodically to store data history
def addData():
   # get the stress tensor (as 3x3 matrix)
   stress=sum(utils.normalShearStressTensors(),Matrix3.Zero)
   # give names to values we are interested in and save them
   plot.addData(exz=O.cell.trsf[0,2],szz=stress[2,2],sxz=stress[0,2],tanPhi=stress[0,2]/stress[2,2],i=O.iter)
   # color particles based on rotation amount
   for b in O.bodies:
      # rot() gives rotation vector between reference and current position
      b.shape.color=utils.scalarOnColorScale(b.state.rot().norm(),0,pi/2.)

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

# better show rotation of particles
Gl1_Sphere.stripes=True

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

O.saveTmp()
```

2.6.5 3d postprocessing

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

# generate loose packing
from yade import pack, qt
sp=pack.SpherePack()
sp.makeCloud((0,0,0),(2,2,2),rMean=.1,rRelFuzz=.6,periodic=True)
# add to scene, make it periodic
sp.toSimulation()
```

```
O.engines=[
   ForceResetter(),
   InsertionSortCollider([Bo1_Sphere_Aabb()]),
   InteractionLoop(
      [Ig2_Sphere_Sphere_L3Geom()],
      [Ip2_FrictMat_FrictMat_FrictPhys()],
      [Law2_L3Geom_FrictPhys_ElPerfPl()]
   ),
   NewtonIntegrator(damping=.4),
   # save data for Paraview
   VTKRecorder(fileName='3d-vtk-',recorders=['all'],iterPeriod=1000),
   # save data from Yade's own 3d view
   qt.SnapshotEngine(fileBase='3d-',iterPeriod=200,label='snapshot'),
   # this engine will be called after 20000 steps, only once
   PyRunner(command='finish()',iterPeriod=20000)
]
O.dt=.5*utils.PWaveTimeStep()

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

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

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

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

2.6.6 Periodic triaxial test

```
# encoding: utf-8

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

```
#

sigmaIso=-1e5

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

# generate loose packing
from yade import pack, qt, plot
sp=pack.SpherePack()
if 0:
   ## uniform distribution
   sp.makeCloud((0,0,0),(2,2,2),rMean=.1,rRelFuzz=.3,periodic=True)
elif 0:
   ## per-fraction distribution
   ## passing: cummulative percentage
   sp.particleSD2(radii=[.09,.1,.2],passing=[40,80,100],periodic=True,numSph=1000)
else:
   ## create packing from clumps
   # configuration of one clump
   c1=pack.SpherePack([((0,0,0),.1),((.15,0,0),.05),((0,.1,0),.05)])
   # make cloud using the configuration c1 (there could c2, c3, ...; selection between them would be random)
   sp.makeClumpCloud((0,0,0),(2,2,2),[c1],periodic=True)

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

O.engines=[
   ForceResetter(),
   InsertionSortCollider([Bo1_Sphere_Aabb()]),
   InteractionLoop(
      [Ig2_Sphere_Sphere_L3Geom()],
      [Ip2_FrictMat_FrictMat_FrictPhys()],
      [Law2_L3Geom_FrictPhys_ElPerfPl()]
   ),
   NewtonIntegrator(damping=.6),
   PeriTriaxController(label='triax',
      # specify target values and whether they are strains or stresses
      goal=(sigmaIso,sigmaIso,sigmaIso),stressMask=7,
      # type of servo-control
      dynCell=True,maxStrainRate=(.1,.1,.1),
      # wait until the unbalanced force goes below this value
      maxUnbalanced=.1,relStressTol=1e-3,
      # call this function when goal is reached and the packing is stable
      doneHook='compactionFinished()'
   ),
   PyRunner(command='addPlotData()',iterPeriod=100),
]
O.dt=.5*utils.PWaveTimeStep()

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

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

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

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

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

Chapter 3

User's manual

3.1 Scene construction

3.1.1 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 forconstructing 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.

Adding multiple particles

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

Many functions introduced in next sections return list of bodies which can be readily added to the simulation, including

- packing generators, such as pack.randomDensePack, pack.regularHexa
- surface function pack.gtsSurface2Facets
- import functions ymport.gmsh, ymport.stl, …

As those functions use 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:

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).

3.1.2 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: mean±dispersion; if dispersion is zero, all spheres will have the same radius.
2. Smooth boundary. Some algorithms treat boundaries in such way that spheres are aligned on them, making them smoother as surface.
3. Packing density, or the ratio of spheres volume and solid size. It is closely related to radius distribution.
4. Coordination number, (average) number of contacts per sphere.
5. Isotropy (related to regularity/irregularity); packings with preferred directions are usually not desirable, unless the modeled solid also has such preference.
6. Permissible Spheres' overlap; some algorithms might create packing where spheres slightly overlap; since overlap usually causes forces in DEM, overlap-free packings are sometimes called "stress-free .

Volume representation

There are 2 methods for representing exact volume of the solid in question in Yade: boundary representation and constructive solid geometry. Despite their fundamental differences, they are abstracted in Yade in the Predicate class. Predicate provides the following functionality:

1. defines axis-aligned bounding box for the associated solid (optionally defines oriented bounding box);
2. can decide whether given point is inside or outside the solid; most predicates can also (exactly or approximately) tell whether the point is inside *and* satisfies some given padding distance from the represented solid boundary (so that sphere of that volume doesn't stick out of the solid).

Constructive Solid Geometry (CSG)

CSG approach describes volume by geometric *primitives* or primitive solids (sphere, cylinder, box, cone, …) and boolean operations on them. Primitives defined in Yade include inCylinder, inSphere, inEllipsoid, inHyperboloid, notInNotch.

For instance, hyperboloid (dogbone) specimen for tension-compression test can be constructed in this way (shown at img. img-hyperboloid):

```
from yade import pack

## construct the predicate first
pred=pack.inHyperboloid(centerBottom=(0,0,-.1),centerTop=(0,0,.1),radius=.05,skirt=.03)
## alternatively: pack.inHyperboloid((0,0,-.1),(0,0,.1),.05,.03)

## pack the predicate with spheres (will be explained later)
spheres=pack.randomDensePack(pred,spheresInCell=2000,radius=3.5e-3)

## add spheres to simulation
O.bodies.append(spheres)
```

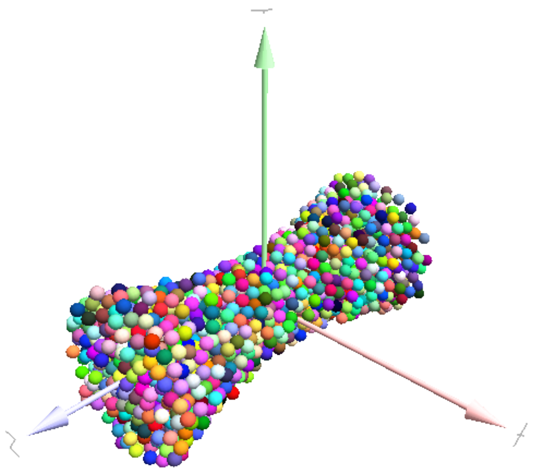


Figure 3.1: Specimen constructed with the pack.inHyperboloid predicate, packed with pack.randomDensePack.

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 π
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))
```

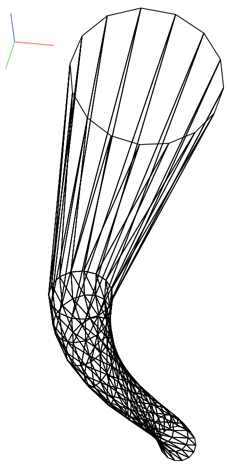


Figure 3.2: Triangulated funnel, constructed with the examples/funnel.py script.

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.

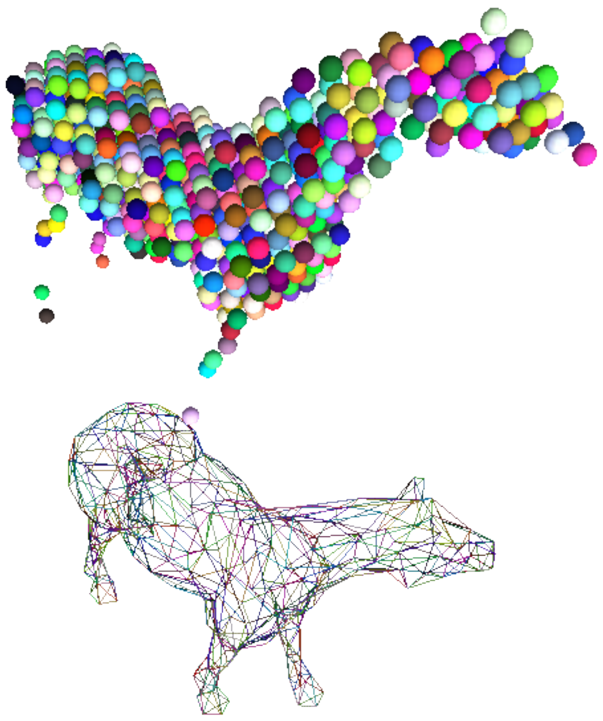


Figure 3.3: Imported GTS surface (horse) used as packing predicate (top) and surface constructed from facets (bottom). See http://www.youtube.com/watch?v=PZVruIlUX1A for movie of this simulation.

Composed predicates also properly define their bounding box. For example, we can take box and remove cylinder from inside, using the `A - B` operation (img. img-predicate-difference):

```
pred=pack.inAlignedBox((-2,-2,-2),(2,2,2))-pack.inCylinder((0,-2,0),(0,2,0),1)
spheres=pack.randomDensePack(pred,spheresInCell=2000,radius=.1,rRelFuzz=.4)
```

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

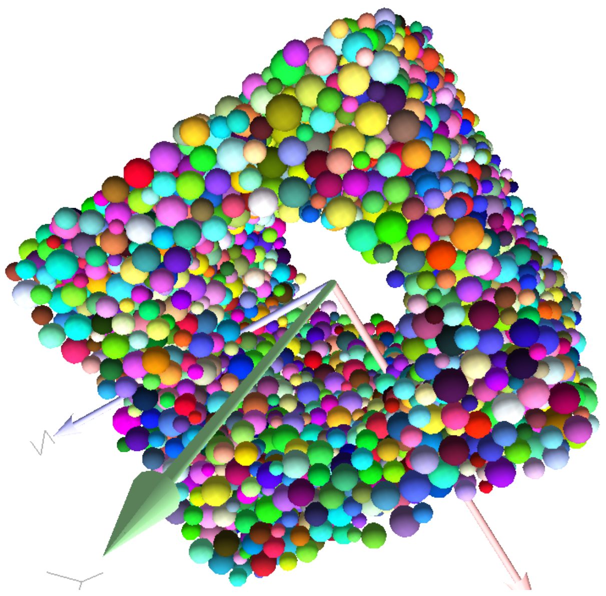


Figure 3.4: Box with cylinder removed from inside, using difference of these two predicates.

used internally for generating initial configuration in dynamic algorithms. For instance:

```
from yade import pack
sp=pack.SpherePack()
sp.makeCloud(minCorner=(0,0,0),maxCorner=(3,3,3),rMean=.2,rRelFuzz=.5)
```

will fill given box with spheres, until no more spheres can be placed. The object can be used to add spheres to simulation:

```
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.

3.1.3 Triangulated surfaces

Yade integrates with the the GNU Triangulated Surface library, exposed in python via the 3rd party gts module. GTS provides variety of functions for surface manipulation (coarsening, tesselation, simplification, import), to be found in its documentation.

GTS surfaces are geometrical objects, which can be inserted into simulation as set of particles whose Body.shape is of type Facet – single triangulation elements. pack.gtsSurface2Facets can be used to convert GTS surface triangulation into list of bodies ready to be inserted into simulation via `O.bodies.append`.

Facet particles are created by default as non-Body.dynamic (they have zero inertial mass). That means that they are fixed in space and will not move if subject to forces. You can however

- prescribe arbitrary movement to facets using a PartialEngine (such as TranslationEngine or RotationEngine);
- assign explicitly mass and inertia to that particle;
- make that particle part of a clump and assign mass and inertia of the clump itself (described below).

Note: Facets can only (currently) interact with spheres, not with other facets, even if they are *dynamic*. Collision of 2 facets will not create interaction, therefore no forces on facets.

Import

Yade currently offers 3 formats for importing triangulated surfaces from external files, in the ymport module:

ymport.gts text file in native GTS format.

ymport.stl STereoLitography format, in either text or binary form; exported from Blender, but from many CAD systems as well.

ymport.gmsh. text file in native format for GMSH, popular open-source meshing program.

If you need to manipulate surfaces before creating list of facets, you can study the py/ymport.py file where the import functions are defined. They are rather simple in most cases.

Parametric construction

The gts module provides convenient way of creating surface by vertices, edges and triangles.

Frequently, though, the surface can be conveniently described as surface between polylines in space. For instance, cylinder is surface between two polygons (closed polylines). The pack.sweptPolylines2gtsSurface offers the functionality of connecting several polylines with triangulation.

Note: The implementation of pack.sweptPolylines2gtsSurface is rather simplistic: all polylines must be of the same length, and they are connected with triangles between points following their indices within each polyline (not by distance). On the other hand, points can be co-incident, if the `threshold` parameter is positive: degenerate triangles with vertices closer that `threshold` are automatically eliminated.

Manipulating lists efficiently (in terms of code length) requires being familiar with list comprehensions in python.

Another examples can be found in examples/mill.py (fully parametrized) or examples/funnel.py (with hardcoded numbers).

3.1.4 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 usepermitting).

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.

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.

3.1.5 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:

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

Functors choice

In the above example, we omited functors, only writing ellipses ... instead. As explained in *Dispatchers and functors*, there are 4 kinds of functors and associated dispatchers. User can choose which ones to use, though the choice must be consistent.

Bo1 functors

`Bo1` functors must be chosen depending on the collider in use; they are given directly to the collider (which internally uses BoundDispatcher).

At this moment (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:

```
O.engines=[...,
   InsertionSortCollider([Bo1_Sphere_Aabb(),Bo1_Facet_Aabb()]),
   ...
]
```

Using more functors than necessary (such as Bo1_Facet_Aabb if there are no facets in the simulation) has no performance penalty. On the other hand, missing functors for existing shapes will cause those bodies to not collider 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 `Ip2` functors; there are no restrictions on its choice in itself, as it only applies forces without creating new objects.

In most simulations, only one `Law2` functor will be in use; it is possible, though, to have several of them, dispatched based on combination of IGeom and IPhys produced previously by `Ig2` and `Ip2` functors respectively (in turn based on combination of Shapes and Materials).

Note: As in the case of `Ip2` functors, receiving a combination of IGeom and IPhys which is not handled by any `Law2` functor is an error.

Examples

Let us give several example 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()]
)
```

3.1.6 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:

  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 expoited 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 ids attribute determining bodies which will be affected. Several of them warrant explicit mention here:

- TranslationEngine and RotationEngine for applying constant speed linear and rotational motion on subscribers.
- ForceEngine and TorqueEngine applying given values of force/torque on subscribed bodies at every step.
- StepDisplacer for applying generalized displacement delta at every timestep; designed for precise control of motion when testing constitutive laws on 2 particles.

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.

3.1.7 Convenience features

Labeling things

Engines and functors can define that `label` attribute. Whenever the `O.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:

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 `=`.

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*:

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.

3.2 Controlling simulation

3.2.1 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`), virutal 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.

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 of trailing whitespaces. They are handled specially in plot.plots and should not be used (see below).

Labels can be conveniently used to access engines in the `myAddData` function:

```
O.engines=[...,
        UniaxialStrainer(...,label='strainer')
]
def myAddData():
        plot.addData(sigma=strainer.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')}
```

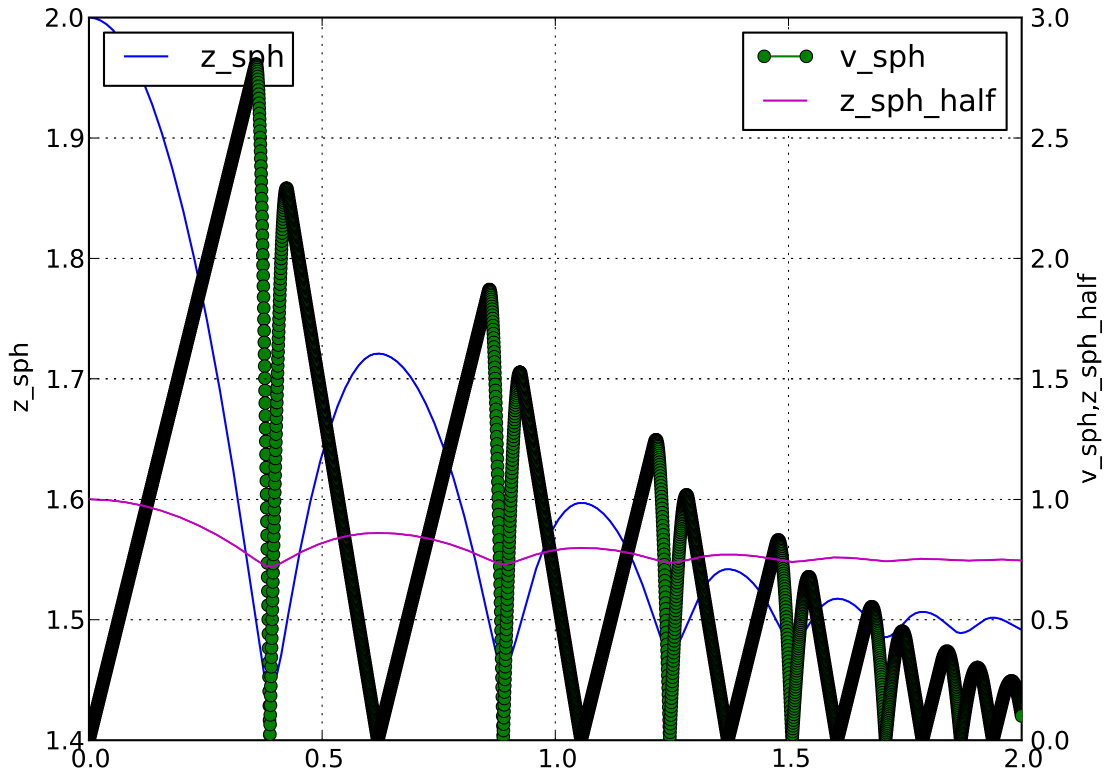


Figure 3.5: Figure generated by examples/simple-scene/simple-scene-plot.py.

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-.'))}
```

And this one will produce a following graph:

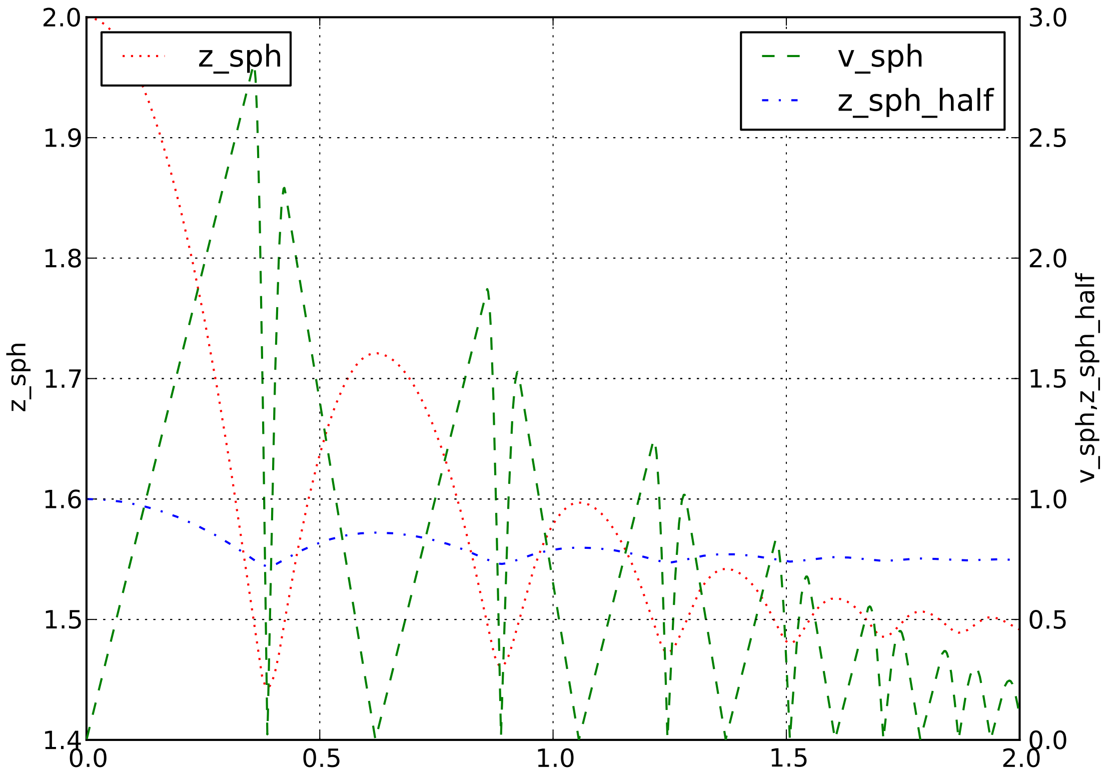


Figure 3.6: Figure generated by changing parameters to plot.plots as above.

```
plot.plots={'i':(('t','xr:'),),'t':(('z_sph','Hr:'),None,('v_sph','+g--'),('z_sph_half','*b-.'))}
```

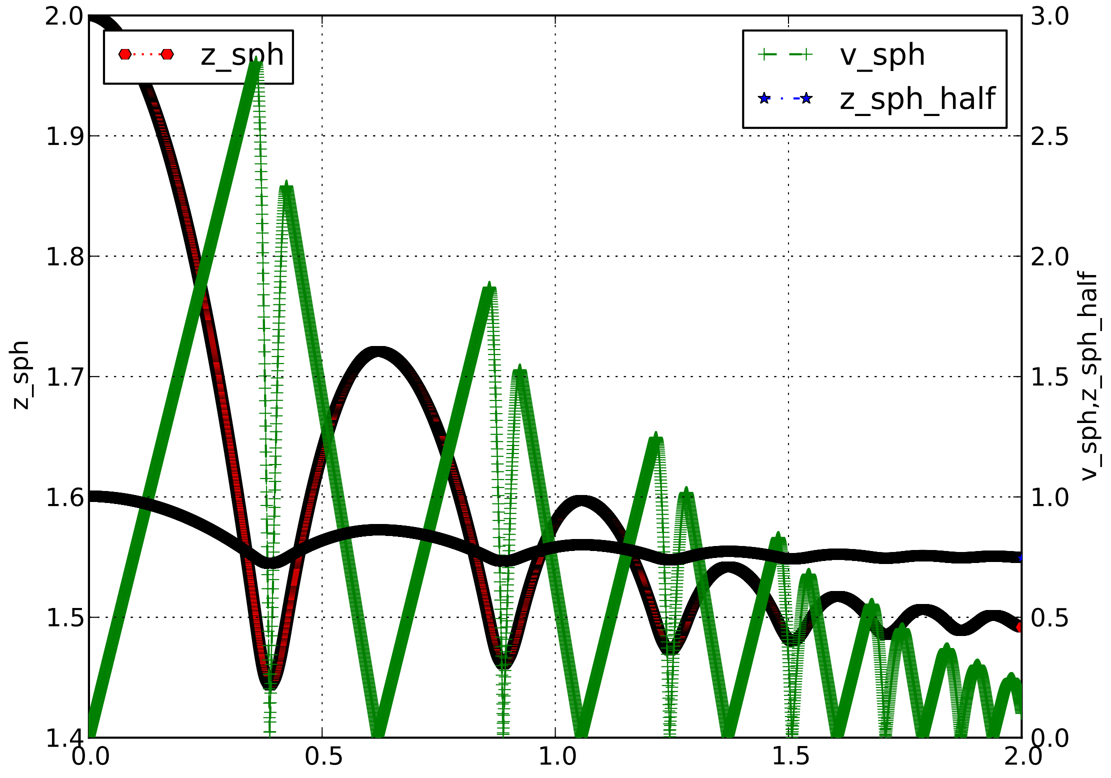


Figure 3.7: Figure generated by changing parameters to plot.plots as above.

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 `'$\alpha$'`, `'$\beta$'` etc. in those labels. To change the font style a following command could be used:

```
yade.plot.matplotlib.rc('mathtext', fontset='stixsans')
```

But this is not part of yade, but a part of matplotlib, and if you want something more complex you really should have a look at matplotlib users manual http://matplotlib.sourceforge.net/users/index.html

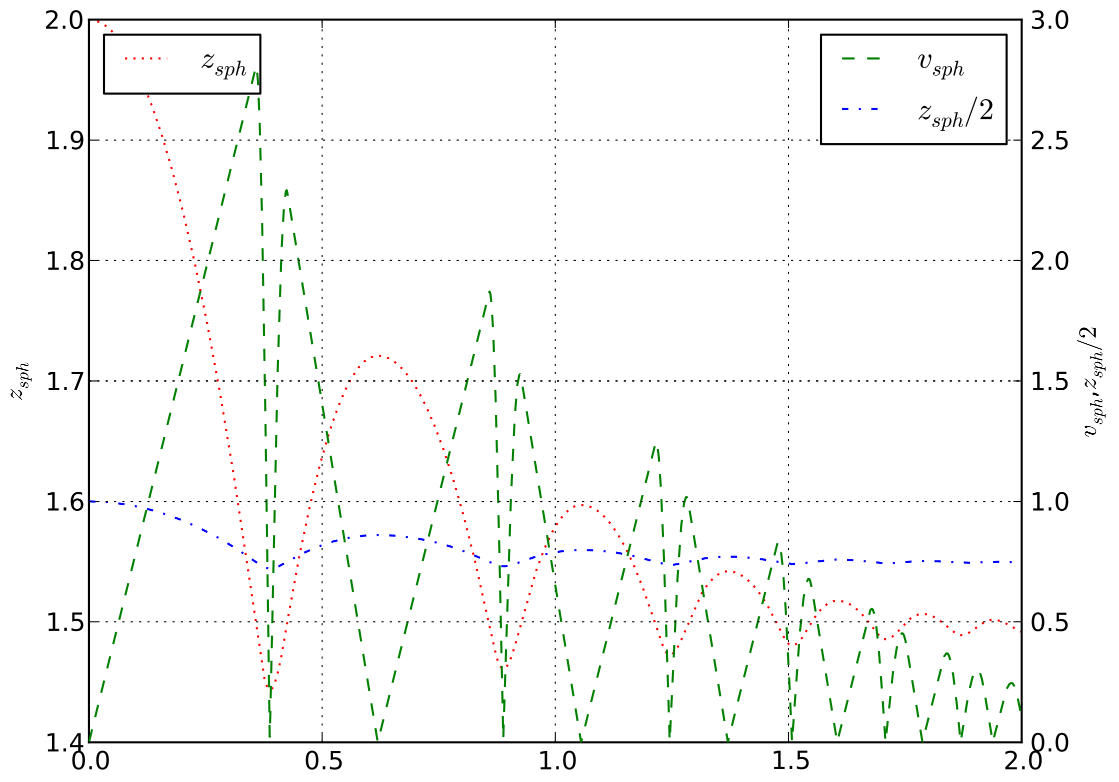


Figure 3.8: Figure generated by examples/simple-scene/simple-scene-plot.py, with TeX labels.

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:

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:

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.

3.2.2 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()
```

3.2.3 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.

3.2.4 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.

Suppsoe we want to study influence of parameters *density* and *initialVelocity* on position of a sphere falling on fixed box. We create parameter table like this:

```
description density initialVelocity # first non-empty line are column headings
reference   2400    10
hi_v           =    20              # = to use value from previous line
lo_v           =     5
# comments are allowed
hi_rho      5000    10
# blank lines as well:

hi_rho_v       =    20
hi_rh0_lo_v    =     5
```

Each line give one combination of these 2 parameters and assigns (which is optional) a *description* of this simulation.

In the simulation file, we read parameters from table, at the beginning of the script; each parameter has default value, which is used if not specified in the parameters file:

```python
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.

```python
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 jo
Will use table `batch.table', with available lines 2, 3, 4, 5, 6, 7.
Will use lines  2 (reference), 3 (hi_v), 4 (lo_v), 5 (hi_rho), 6 (hi_rho_v), 7 (hi_rh0_lo_v).
Master process pid 7030
```

These lines inform us about general batch information: nice level, log file names, how many cores will be used (4); table name, and line numbers that contain parameters; finally, which lines will be used; master PID is useful for killing (stopping) the whole batch with the `kill` command.

```
Job summary:
   #0 (reference/4): PARAM_TABLE=batch.table:2 DISPLAY=  /usr/local/bin/yade-trunk --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 batc
   #5 (hi_rh0_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_rh0_lo_v/4) started on Tue Apr 13 13:59:40 2010
#5 (hi_rh0_lo_v/4) done    (exit status 0), duration 00:00:01, log batch.hi_rh0_lo_v.log
```

information about job status changes is being printed, until:

```
All jobs finished, total time  00:00:08
Log files:
batch.reference.log batch.hi_v.log batch.lo_v.log batch.hi_rho.log batch.hi_rho_v.log batch.hi_rh0_lo_v.log
Bye.
```

Separating output files from jobs

As one might output data to external files during simulation (using classes such as VTKRecorder, it is important to name files in such way that they are not overwritten by next (or concurrent) job in the same batch. A special tag `O.tags['id']` is provided for such purposes: it is comprised of date, time and PID, which makes it always unique (e.g. `20100413T144723p7625`); additional advantage is that alphabetical order of the `id` tag is also chronological.

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 compuation

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 heiristic 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.

HTTP overview

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

3.3 Postprocessing

3.3.1 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.

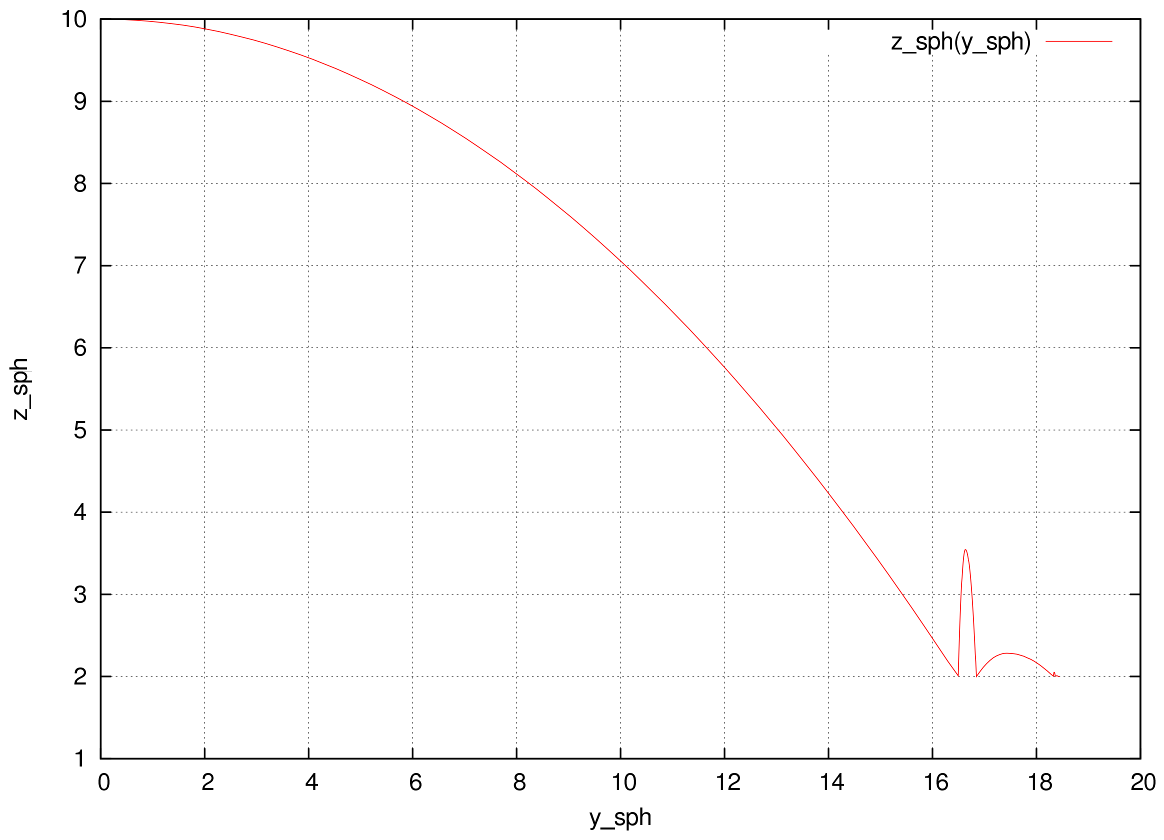


Figure 3.9: Figure from single job in the batch.

2. Periodic frame snapshot using SnapshotEngine (see examples/bulldozer.py for a full example):

   ```
   O.engines=[
     #...
     SnapshotEngine(iterPeriod=100,fileBase='/tmp/bulldozer-',viewNo=0,label='snapshooter')
   ]
   ```

   which will save numbered files like `/tmp/bulldozer-0000.png`. These files can be processed externally (with mencoder and similar tools) or directly with the utils.makeVideo:

   ```
   utils.makeVideo(frameSpec,out,renameNotOverwrite=True,fps=24,kbps=6000,bps=None)
   ```

   The video is encoded using the default mencoder codec (mpeg4).

3. Specialized post-processing tools, notably Paraview. This is described in more detail in the following section.

Paraview

Saving data during the simulation

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

```
O.engines=[
        # ...
        VTKRecorder(iterPeriod=100,recorders=['spheres','facets','colors'],fileName='/tmp/p1-')
]
```

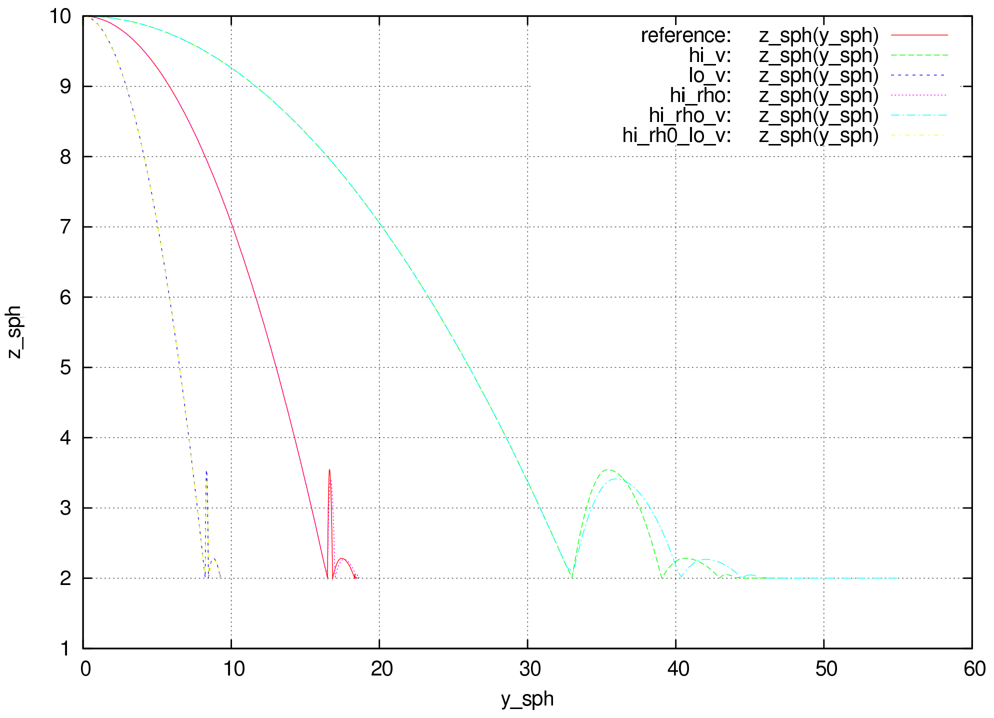
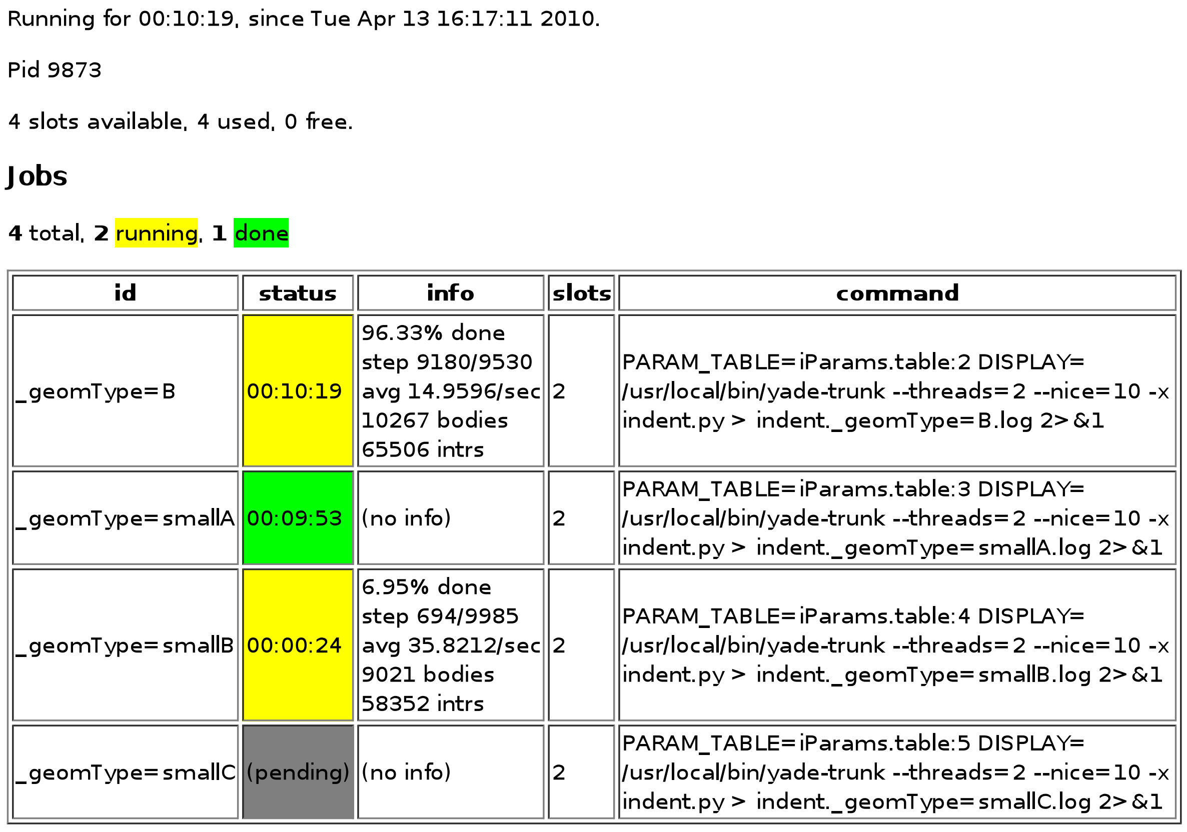


Figure 3.10: Merged figure from all jobs in the batch. Note that labels are prepended by job description to make lines distinguishable.



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

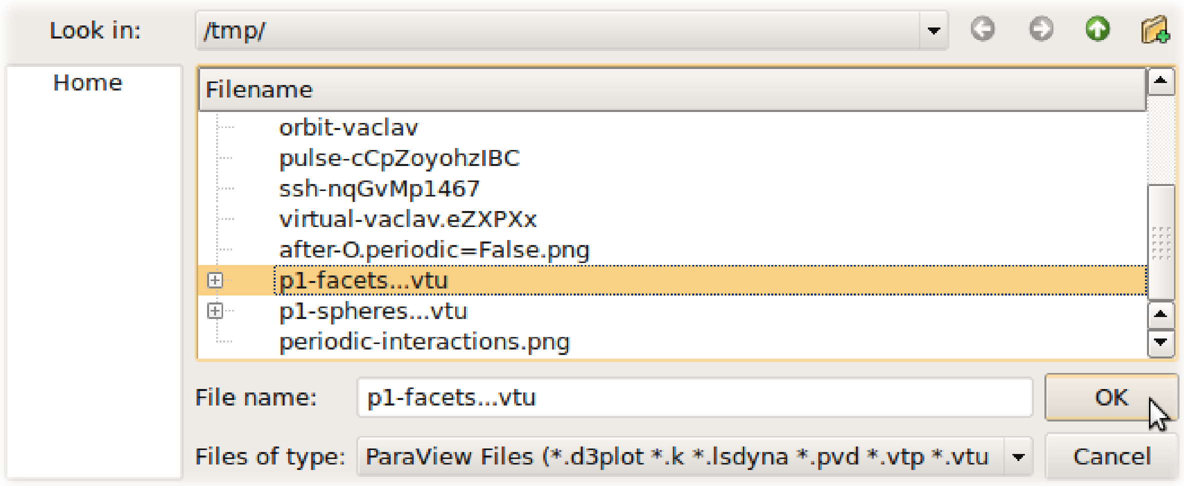
id	status	info	slots	command
_geomType=B	00:10:19	96.33% done step 9180/9530 avg 14.9596/sec 10267 bodies 65506 intrs	2	PARAM_TABLE=iParams.table:2 DISPLAY= /usr/local/bin/yade-trunk --threads=2 --nice=10 -x indent.py > indent._geomType=B.log 2>&1
_geomType=smallA	00:09:53	(no info)	2	PARAM_TABLE=iParams.table:3 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 3.11: Summary page available at port 9080 as batch is processed (updates every 5 seconds automatically). Possible job statuses are pending, running, done, failed.

- 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:



Look in: /tmp/

Home

Filename
orbit-vaclav
pulse-cCpZoyohzIBC
ssh-nqGvMp1467
virtual-vaclav.eZXPXx
after-O.periodic=False.png
p1-facets...vtu
p1-spheres...vtu
periodic-interactions.png

File name: p1-facets...vtu

Files of type: ParaView Files (*.d3plot *.k *.lsdyna *.pvd *.vtp *.vtu

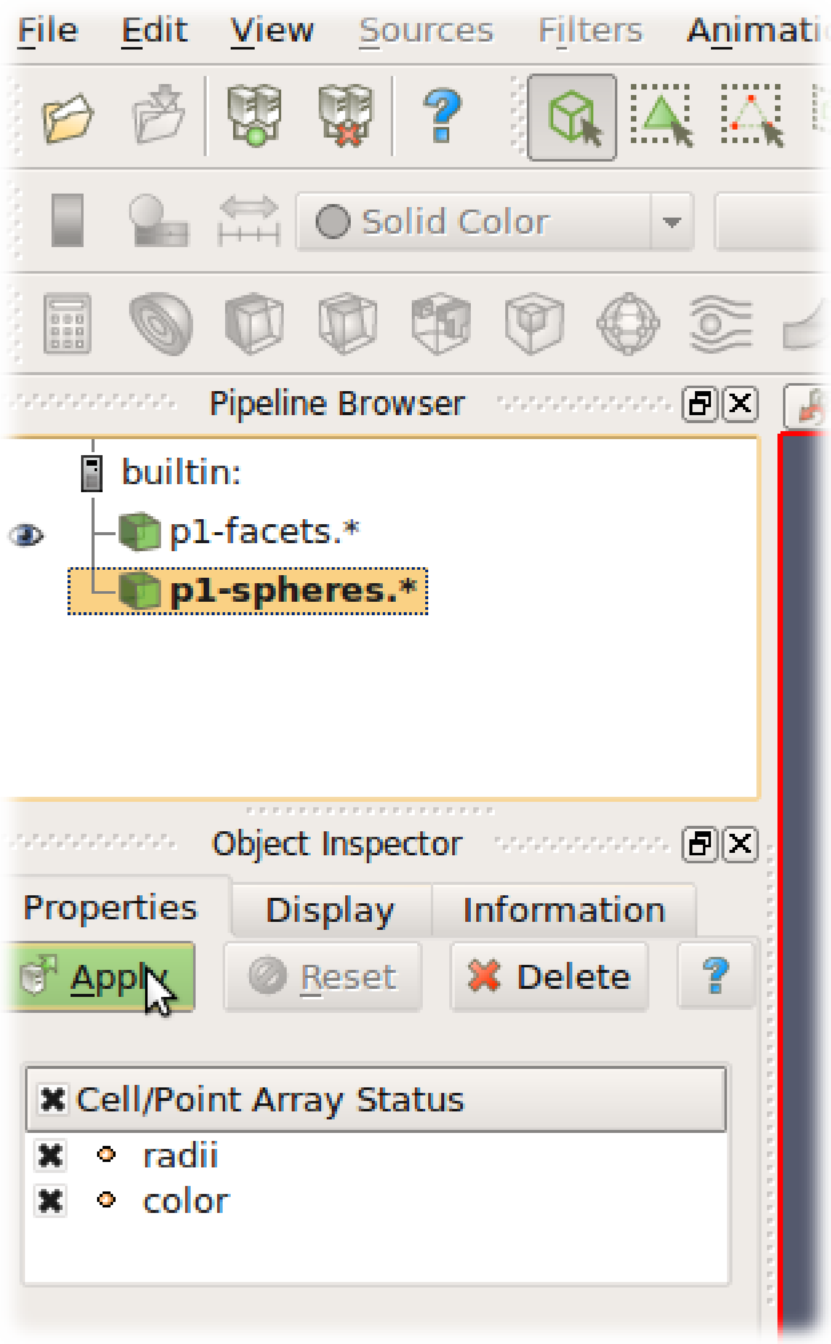
Click on the "Apply" button in the "Object inspector" sub-window to make loaded objects visible. You can see tree of displayed objects in the "Pipeline browser":

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.



Cell/Point Array Status
radii
color

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*.

3.4 Python specialties and tricks

3.5 Extending Yade

- new particle shape
- new constitutive law

3.6 Troubleshooting

3.6.1 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 your particular simulation (such as impossible dispatch);
- problem with graphics card driver.

Try to reproduce the error (run the same script) with debug-enabled version of Yade. Debugger will be automatically launched at crash, showing backtrace of the code (in this case, we triggered crash by hand):

```
Yade [1]: import os,signal
Yade [2]: os.kill(os.getpid(),signal.SIGSEGV)
SIGSEGV/SIGABRT handler called; gdb batch file is `/tmp/yade-YwtfRY/tmp-0'
GNU gdb (GDB) 7.1-ubuntu
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
[Thread debugging using libthread_db enabled]
[New Thread 0x7f0fb1268710 (LWP 16471)]
[New Thread 0x7f0fb29f2710 (LWP 16470)]
[New Thread 0x7f0fb31f3710 (LWP 16469)]
```

…

What looks as cryptic message is valuable information for developers to locate source of the bug. In particular, there is (usually) line `<signal handler called>`; lines below it are source of the bug (at least very likely so):

```
Thread 1 (Thread 0x7f0fcee53700 (LWP 16465)):
#0  0x00007f0fcd8f4f7d in __libc_waitpid (pid=16497, stat_loc=<value optimized out>, options=0) at ../sysdeps/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  0x00000000004a7c5e 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.

3.6.2 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.

3.6.3 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 4

Programmer's manual

4.1 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.

4.1.1 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 reffered 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` [**-1, which means that the opposite of `debug` value is used**] 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 preprocessor (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 are 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` [**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 is the scripting language used by yade. Besides [boost::python]_, yade further requires

- ipython (terminal interaction)
- matplotlib (plotting)
- numpy (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.

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 ses its routines.

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

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.

4.1.2 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 `SConscript`, 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:

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

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 joind 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:

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

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 345uiysdijkn);
```

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.	`core`, `yade-support`; may **not** depend on external libraries explcitly (only implicitly, by adding the library to global linker flags in `SConstruct`)
extra	(undefined)	(arbitrary)
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 scannig their `#include` directives for the pattern `#include<yade/module/Plugin.hpp>`. Again, this works well in normal circumastances, but is not necessarily robust.

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

4.2 Development tools

4.2.1 Integrated Development Environment and other tools

A frequently used IDE is Kdevelop. We recommend using this software for navigating in the sources, compiling and debugging. Other useful tools for debugging and profiling are Valgrind and KCachegrind. A series a wiki pages are dedicated to these tools in the development section of the wiki.

4.2.2 Hosting and versioning

The Yade project is kindly hosted at launchpad, which is used for source code, bug tracking, planning, package downloads and more. Our repository can be http-browsed.

The versioning software used is Bazaar, for which a short tutorial can be found in a Yade's wiki pages. Bazaar is a distributed revision control system. It is available packaged for all major linux distributions.

4.2.3 Build robot

A build robot hosted at 3SR lab. is tracking souce code changes. Each time a change in the source code is commited to the main development branch via bazaar, the "buildbot" downloads and compiles the new version, and start a series of tests.

If a compilation error has been introduced, it will be notified to the yade-dev mailing list and to the commiter, thus helping to fix problems quickly. If the compilation is successfull, the buildbot starts unit regression tests and "check tests" (see below) and report the results. If all tests are passed, a new version of the documentation is generated and uploaded to the website in html and pdf format. As a consequence, those two links always point to the documentation (the one you are reading now) of the last successfull build, and the delay between commits and documentation updates are very short (minutes). The buildbot activity and logs can be browsed online.

4.2.4 Regression tests

Yade contains two types of regression tests, some are unit tests while others are testing more complex simulations. Altough both types can be considered regression tests, the usage is that we name the first simply "regression tests", while the latest are called "check tests". Both series of tests can be ran at yade startup by passing the options "test" or "check"

```
yade --test
yade --check
```

Unit regression tests

Unit regression tests are testing the output of individual functors and engines in well defined conditions. They are defined in the folder py/tests/. The purpose of unit testing is to make sure that the behaviour of the most important classes remains correct during code development. Since they test classes one by one, unit tests can't detect problems coming from the interaction between different engines in a typical simulation. That is why check tests have been introduced.

Check tests

Check tests perform comparisons of simulation results between different versions of yade, as discussed in http://www.mail-archive.com/yade-dev@lists.launchpad.net/msg05784.html and the whole thread. They differ with regression tests in the sense that they simulate more complex situations and combinations of different engines, and usually don't have a mathematical proof (though there is no restriction on the latest). They compare the values obtained in version N with values obtained in a previous version or any other "expected" results. The reference values must be hardcoded in the script itself or in data files provided with the script. Check tests are based on regular yade scripts, so that users can easily commit their own scripts to trunk in order to get some automatized testing after commits from other developers.

Since the check tests history will be mostly based on standard output generated by "yade —check", a meaningfull checkTest should include some "print" command telling if something went wrong. If the script itself fails for some reason and can't generate an output, the log will contain "scriptName failure". If the script defines differences on obtained and awaited data, it should print some useful information about the problem and increase the value of global variable resultStatus. After this occurs, the automatic test will stop the execution with error message.

An example check test can be found in checkTestTriax.py. It shows results comparison, output, and how to define the path to data files using "checksPath". Users are encouraged to add their own scripts into the scripts/test/checks/ folder. Discussion of some specific checktests design in users question is welcome. Note that re-compiling is required before that added scripts can be launched by "yade —check" (or direct changes have to be performed in "lib" subfolders). A check test should never need more than a few seconds to run. If your typical script needs more, try and reduce the number of element or the number of steps.

4.3 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; inadverently 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 ouput 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 unitialized variable. (This is taken care of by *YADE_CLASS_BASE_DOC_* macro family* macros for user classes)

4.3.1 Class naming

Although for historial 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 dispatchers [2] are defined:

dispatcher	arity	dispatch types	created type	functor type	functor prefix
BoundDispatcher	1	Shape	Bound	BoundFunctor	`Bo1`
IGeomDispatcher	2 (symetric)	2 × Shape	IGeom	IGeomFunctor	`Ig2`
IPhysDispatcher	2 (symetric)	2 × Material	IPhys	IPhysFunctor	`Ip2`
LawDispatcher	2 (asymetric)	IGeom IPhys	*(none)*	LawFunctor	`Law2`

Respective abstract functors for each dispatchers are BoundFunctor, IGeomFunctor, IPhysFunctor and LawFunctor.

Functors Functor name is composed of 3 parts, separated by underscore.

1. prefix, composed of abbreviated functor type and arity (see table above)
2. Types entering the dispatcher logic (1 for unary and 2 for binary functors)
3. Return type for functors that create instances, simple characteristics for functors that don't create instances.

To give a few examples:

- Bo1_Sphere_Aabb is a BoundFunctor which is called for Sphere, creating an instance of Aabb.
- Ig2_Facet_Sphere_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.

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

4.3.2 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)
- 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 used, 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 integration scheme …

Separate class/function documentation

Some c++ might have long or content-rich documentation, which is rather inconvenient to type in the c++ source itself as string literals. Yade provides a way to write documentation separately in py/_extraDocs.py file: it is executed after loading c++ plugins and can set `__doc__` attribute of any object directly, overwriting docstring from c++. In such (exceptional) cases:

1. Provide at least a brief description of the class in the c++ code nevertheless, for people only reading the code.
2. Add notice saying "This class is documented in detail in the py/_extraDocs.py file".
3. Add documentation to py/_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 lie 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 bzr, 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.

4.4 Support framework

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

4.4.1 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.

[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.

`YADE_PTR_CAST` expands to `static_pointer_cast` in optimized builds and to `dynamic_pointer_cast` in debug builds.

4.4.2 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`
- `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 firther 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.

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.

4.4.3 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 getClassName()` returning class name (`Foo`) as string. (There is the `typeid(instanceOrType).name()` standard c++ construct, but the name returned is compiler-dependent.)
- `unsigned getBaseClassNumber()` returning number of base classes (in this case, 2).
- `std::string getBaseClassName(unsigned i=0)` returning name of *i*-th base class (here, `Bar` for i=0 and `Baz` for i=1).

Warning: RTTI relies on virtual functions; in order for virtual functions to work, at least one virtual method must be present in the implementation (`.cpp`) file. Otherwise, virtual method table (vtable) will not be generated for this class by the compiler, preventing virtual methods from functioning properly.

[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.

Some RTTI information can be accessed from python:

4.4.4 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 serialzed along with their literal names, by which they are identified. This method then invokes `registerAttributes` of its base class (until `Serializable` itself is reached); in this way, derived classes properly serialize data of their base classes.

  This funcionality is hidden behind the macro *YADE_CLASS_BASE_DOC_* macro family* used in class declaration body (header file), which takes base class and list of attributes:

  ```
  YADE_CLASS_BASE_DOC_ATTRS(ThisClass,BaseClass,"class documentation",((type1,attribute1,initValue1,,"Docume
  ```

  Note that attributes are encodes in double parentheses, not separated by commas. Empty attribute list can be given simply by `YADE_CLASS_BASE_DOC_ATTRS(ThisClass,BaseClass,"documentation",)` (the last comma is mandatory), or by omiting `ATTRS` from macro name and last parameter altogether.

- Fundamental type: strings, various number types, booleans, `Vector3r` and others. Their "handlers" (serializers and deserializers) are defined in `lib/serialization`.
- Standard container of any serializable objects.
- Shared pointer to serializable object.

Yade uses the excellent boost::serialization library internally for serialization of data.

Note: `YADE_CLASS_BASE_DOC_ATTRS` also generates code for attribute access from python; this will be discussed later. Since this macro serves both purposes, the consequence is that attributes that are serialized can always be accessed from python.

Yade also provides callback for before/after (de) serialization, virtual functions Serializable::preProcessAttributes and Serializable::postProcessAttributes, which receive one `bool deserializing` argument (`true` when deserializing, `false` when serializing). Their default implementation in Serializable doesn't do anything, but their typical use is:

- converting some non-serializable internal data structure of the class (such as multi-dimensional array, hash table, array of pointers) into a serializable one (pre-processing) and fill this non-serializable structure back after deserialization (post-processing); for instance, InteractionContainer uses these hooks to ask its concrete implementation to store its contents to a unified storage (`vector<shared_ptr<Interaction> >`) before serialization and to restore from it after deserialization.
- precomputing non-serialized attributes from the serialized values; e.g. Facet computes its (local) edge normals and edge lengths from vertices' coordinates.

Class factory

Each serializable class must use `REGISTER_SERIALIZABLE`, which defines function to create that class by `ClassFactory`. `ClassFactory` is able to instantiate a class given its name (as string), which is necessary for deserialization.

Although mostly used internally by the serialization framework, programmer can ask for a class instantiation using `shared_ptr<Factorable> f=ClassFactory::instance().createShared("ClassName");`, casting the returned `shared_ptr<Factorable>` to desired type afterwards. Serializable itself derives from `Factorable`, i.e. all serializable types are also factorable (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 place in the implementation (.cpp) file. It simple 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 attributes makes the function being executed when the plugin is loaded via `dlopen` from `ClassFactory::load(...)`. It registers all factorable classes from that file in the *Class factory*.

Note: Classes that do not derive from `Factorable`, such as `Shop` or `SpherePack`, are not declared with `YADE_PLUGIN`.

This is an example of a serializable class header:

```
/*! Homogeneous gravity field; applies gravity×mass force on all bodies. */
class GravityEngine: public GlobalEngine{
        public:
                virtual void action();
        // registering class and its base for the RTTI system
        YADE_CLASS_BASE_DOC_ATTRS(GravityEngine,GlobalEngine,
                // documentation visible from python and generated reference documentation
                "Homogeneous gravity field; applies gravity×mass force on all bodies.",
                // enumerating attributes here, include documentation
                ((Vector3r,gravity,Vector3r::ZERO,"acceleration, zero by default [kgms ²]"))
        );
};
// registration function for ClassFactory
REGISTER_SERIALIZABLE(GravityEngine);
```

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):

and the XML looks like this:

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:

4.4.5 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 parethesized list:

```
((type1,attr1,initValue1,attrFlags,"Attribute 1 documentation"))
((type2,attr2,,,"Attribute 2 documentation"))  // initValue and attrFlags unspecified
```

This will expand to

1. data members declaration in c++ (note that all attributes are *public*):

```
public: type1 attr1;
        type2 attr2;
```

2. Initializers of the default (argument-less) constructor, for attributes that have non-empty `initValue`:

```
Klass(): attr1(initValue1), attr2() { /* constructor body */ }
```

No initial value will be assigned for attribute of which initial value is left empty (as is for attr2 in the above example). Note that you still have to write the commas.

3. Registration of the attribute in the serialization system (unless disabled by attrFlags – see below)

4. **Registration of the attribute in python (unless disabled by attrFlags), so that it can be accesse**
   The attribute is read-write by default, see attrFlags to change that.

   This attribute will carry the docstring provided, along with knowledge of the initial value. You can add text description to the default value using the comma operator of c++ and casting the char* to (void):

```
((Real,dmgTau,((void)"deactivated if negative",-1),,"Characteristic time for normal viscosity. [s
```

leading to CpmMat::dmgTau.

The attribute is registered via `boost::python::add_property` specifying `return_by_value` policy rather than `return_internal_reference`, which is the default when using `def_readwrite`. The reason is that we need to honor custom converters for those values; see note in *Custom converters* for details.

Attribute flags

By default, an attribute will be serialized and will be read-write from python. There is a number of flags that can be passed as the 4th argument (empty by default) to change that:

- `Attr::noSave` avoids serialization of the attribute (while still keeping its accessibility from Python)
- `Attr::readonly` makes the attribute read-only from Python

- `Attr::triggerPostLoad` will trigger call to `postLoad` function to handle attribute change after its value is set from Python; this is to ensure consistency of other precomputed data which depend on this value (such as `Cell.trsf` and such)
- `Attr::hidden` will not expose the attribute to Python at all
- `Attr::noResize` will not permit changing size of the array from Python [not yet used]

Flags can be combined as usual using bitwise disjunction `|` (such as `Attr::noSave | Attr::readonly`), though in such case the value should be parenthesized to avoid a warning with some compilers (g++ specifically), i.e. `(Attr::noSave | Attr::readonly)`.

Currently, the flags logic handled at runtime; that means that even for attributes with `Attr::noSave`, their serialization template must be defined (although it will never be used). In the future, the implementation might be template-based, avoiding this necessity.

`deprec` List of deprecated attribute names. The syntax is

```
((oldName1,newName1,"Explanation why renamed etc."))
((oldName2,newName2,"! Explanation why removed and what to do instaed."))
```

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 straightforwardsly replaced by another, but more complex adaptation of user's script is needed. You still have to give `newName2`, although its value will never be used – you can use any variable you like, but something must be given for syntax reasons).

Warning: Due to compiler limitations, this feature only works if Yade is compiled with gcc >= 4.4. In the contrary case, deprecated attribute functionality is disabled, even if such attributes are declared.

`init` Parethesized list of the form:

```
((attr3,value3)) ((attr4,value4))
```

which will be expanded to initializers in the default ctor:

```
Klass(): /* attributes declared with the attrs argument */ attr4(value4), attr5(value5) { /* constructor b
```

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 ouside 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 appeneded 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 in**

    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++.

4.4.6 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. Numer 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 are 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	BoundFunctor, IGeomDispatcher
Material	IPhysDispatcher
IPhys	LawDispatcher
IGeom	LawDispatcher

The top-level Indexable must use the `REGISTER_INDEX_COUNTER` macro, which sets up the machinery for identifying types of derived classes; they must then use the `REGISTER_CLASS_INDEX` macro *and* call `createIndex()` in their constructor. For instance, taking the Shape class (which is a top-level Indexable):

```
// derive from Indexable
class Shape: public Serializable, public Indexable {
   // never call createIndex() in the top-level Indexable ctor!
   /* … */

   // allow index registration for classes deriving from ``Shape``
   REGISTER_INDEX_COUNTER(Shape);
};
```

Now, all derived classes (such as Sphere or Facet) use this:

```
class Sphere: public Shape{
   /* … */
   YADE_CLASS_BASE_DOC_ATTRS_CTOR(Sphere,Shape,"docstring",
      ((Type1,attr1,default1,"docstring1"))
      /* … */,
      // this is the CTOR argument
         // important; assigns index to the class at runtime
         createIndex();
   );
   // register index for this class, and give name of the immediate parent class
   //    (i.e. if there were a class deriving from Sphere, it would use
   //     REGISTER_CLASS_INDEX(SpecialSphere,Sphere),
   //     not REGISTER_CLASS_INDEX(SpecialSphere,Shape)!)
   REGISTER_CLASS_INDEX(Sphere,Shape);
};
```

[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.

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):

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:

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

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):

OpenGL functors

OpenGL rendering is being done also by 1D functors (dispatched for the type to be rendered). Since it is sufficient to have exactly one class for each rendered type, the functors are found automatically. Their base functor types are `GlShapeFunctor`, `GlBoundFunctor`, `GlIGeomFunctor` and so on. These classes register the type they render using the `RENDERS` macro:

```
class Gl1_Sphere: public GlShapeFunctor {
   public :
      virtual void go(const shared_ptr<Shape>&,
         const shared_ptr<State>&,
         bool wire,
         const GLViewInfo&
      );
   RENDERS(Sphere);
   YADE_CLASS_BASE_DOC_STATICATTRS(Gl1_Sphere,GlShapeFunctor,"docstring",
      ((Type1,staticAttr1,informativeDefault,"docstring"))
      /* … */
   );
};
REGISTER_SERIALIZABLE(Gl1_Sphere);
```

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

Note: OpenGL functors may disappear in the future, being replaced by virtual functions of each class that can be rendered.

4.4.7 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-instrusive way of parallelizing was used: OpenMP. OpenMP is standartized shared-memory parallel execution environment, where parallel sections are marked by special `#pragma` in the code (which means that they can compile with compiler that doesn't support OpenMP) and a few functions to query/manipulate OpenMP runtime if necessary.

There is parallelism at 3 levels:

- Computation, interaction (python, GUI) and rendering threads are separate. This is done via regular threads (boost::threads) and is not related to OpenMP.
- ParallelEngine can run multiple engine groups (which are themselves run serially) in parallel; it rarely finds use in regular simulations, but it could be used for example when coupling with an independent expensive computation:

  ```
  ParallelEngine([
          [Engine1(),Engine2()],   # Engine1 will run before Engine2
          [Engine3()]              # Engine3() will run in parallel with the group [Engine1(),Engine2()
                                   # arbitrary number of groups can be used
  ])
  ```

  `Engine2` will be run after `Engine1`, but in parallel with `Engine3`.

  Warning: It is your reponsibility to avoid concurrent access to data when using ParallelEngine. Make sure you understand *very well* what the engines run in parallel do.

- Parallelism inside Engines. Some loops over bodies or interactions are parallelized (notably InteractionLoop and NewtonIntegrator, which are treated in detail later (FIXME: link)):

  ```
  #pragma omp parallel for
  for(long id=0; id<size; id++){
     const shared_ptr<Body>& b(scene->bodies[id]);
     /* … */
  }
  ```

  Note: OpenMP requires loops over contiguous range of integers (OpenMP 3 also accepts containers with random-access iterators).

  If you consider running parallelized loop in your engine, always evalue its benefits. OpenMP has some overhead fo creating threads and distributing workload, which is proportionally more expensive if the loop body execution is fast. The results are highly hardware-dependent (CPU caches, RAM controller).

Maximum number of OpenMP threads is determined by the `OMP_NUM_THREADS` environment variable and is constant throughout the program run. Yade main program also sets this variable (before loading OpenMP libraries) if you use the `-j`/`--threads` option. It can be queried at runtime with the `omp_get_max_threads` function.

At places which are susceptible of being accessed concurrently from multiple threads, Yade provides some mutual exclusion mechanisms, discussed elsewhere (FIXME):

- simultaneously writeable container for *ForceContainer*,
- mutex for Body::state.

4.4.8 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 uninsteresting 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).

[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.

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 be prec
  ```

  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 newtonRaphson
  ```

  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):

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

4.4.9 Timing

Yade provides 2 services for measuring time spent in different pars 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 numebr 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:

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 cummulatively time differences between *checkpoints*.

Note: Fine timing with `TimingDeltas` will only work if timing is enabled globally (see previous section). The code would still run, but giving zero times and exec counts.

1. Engine::timingDeltas must point to an instance of TimingDeltas (prefferably 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 disproportionally longer time that the rest of engine; in the output presented above, LawDispatcher takes almost of total simulation time in average, but the number would be twice of thrice lower typically (note that each checkpoint was timed almost 5 million times in this particular case).

4.4.10 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*.

`Gl1_*` functors operating on Body's attributes (Shape, State, Bound) are called with the OpenGL context translated and rotated according to State::pos and State::ori. Interaction functors work in global coordinates.

4.5 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.

4.5.1 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 interally a `Scene* scene` pointer which is updated regularly by engine/functor callers; this ensures that the current scene can be accessed from within user code.

For outside functions (such as those called from python, or static functions in `Shop`), you can use `Omega::instance().getScene()` to retrieve a `shared_ptr<Scene>` of the current scene.

4.5.2 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…n indexability),
- store `shared_ptr`, not objects themselves,
- *no* mutual exclusion for insertion/removal (this must be assured by the called, if desired),
- intelligent allocation of `id` for new bodies (tracking removed bodies),
- easy iteration over all bodies.

Note: Currently, there is "abstract" class `BodyContainer`, from which derive concrete implementations; the initial idea was the ability to select at runtime which implementation to use (to find one that performs the best for given simulation). This incurs the penalty of many virtual function calls, and will probably change in the future. All implementations of BodyContainer were removed in the meantime, except `BodyVector` (internally a `vector<shared_ptr<Body> >` plus a few methods around), which is the fastest.

Insertion/deletion

Body insertion is typically used in FileGenerator's:

```
shared_ptr<Body> body(new Body);
// ... (body setup)
scene->bodies->insert(body); // assigns the id
```

Bodies are deleted only rarely:

```
scene->bodies->erase(id);
```

Warning: Since mutual exclusion is not assured, never insert/erase bodies from parallel sections, unless you explicitly assure there will be no concurrent access.

Iteration

The container can be iterated over using `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 as silently skipped:

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

4.5.3 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 eleents of simulation must be synchronized; the exact workflow is described in *Handling interactions*. You will almost certainly never need to insert/delete an interaction manually from the container; if you do, consider designing your code differently.

```
// both insertion and erase are internally protected by a mutex,
// and can be done from parallel sections safely
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(!->isReal()) continue;
   /* … */
}
```

4.5.4 ForceContainer

ForceContainer holds "generalized forces", i.e. forces, torques, (explicit) dispalcements and rotations for each body.

During each computation step, there are typically 3 phases pertaining to forces:

1. Resetting forces to zero (usually done by the ForceResetter engine)
2. Incrementing forces from parallel sections (solving interactions – from LawFunctor)
3. Reading absolute force values sequentially for each body: forces applied from different interactions are summed together to give overall force applied on that body (NewtonIntegrator, but also various other engine that read forces)

This scenario leads to special design, which allows fast parallel write access:

- each thread has its own storage (zeroed upon request), and only writes to its own storage; this avoids concurrency issues. Each thread identifies itself by the omp_get_thread_num() function provided by the OpenMP runtime.
- before reading absolute values, the container must be synchronized, i.e. values from all threads are summed up and stored separately. This is a relatively slow operation and we provide ForceContainer::syncCount that you might check to find cummulative number of synchronizations and compare it against number of steps. Ideally, ForceContainer is only synchronized once at each step.
- the container is resized whenever an element outside the current range is read/written to (the read returns zero in that case); this avoids the necessity of tracking number of bodies, but also is potential danger (such as `scene->forces.getForce(1000000000)`, which will probably exhaust your RAM). Unlike c++, Python does check given id against number of bodies.

```
// resetting forces (inside ForceResetter)
scene->forces.reset()

// in a parallel section
scene->forces.addForce(id,force); // add force

// container is not synced after we wrote to it, sync before reading
scene->forces.sync();
const Vector3r& f=scene->forces.getForce(id);
```

Synchronization is handled automatically if values are read from python:

4.5.5 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 stillremain potential-only

   `true` if there is geometrical overlap; the functor will have created an IGeom in such case.

> **Note:** For *real* interactions, the functor *must* return `true`, even if there is no more spatial overlap between bodies. If you wish to delete an interaction without geometrical overlap, you have to do this in the LawFunctor.
>
> This behavior is deliberate, since different laws have different requirements, though ideally using relatively small number of generally useful geometry functors.

> **Note:** If there is no functor suitable to handle given combination of shapes, the interaction will be left in potential state, without raising any error.

4. For real interactions (already existing or just created in last step), InteractionLoop (via IPhysDispatcher) calls appropriate IPhysFunctor based on Material combination of both bodies. The functor *must* update (or create, if it doesn't exist yet) associated IPhys instance. It is an error if no suitable functor is found, and an exception will be thrown.
5. For real interactions, InteractionLoop (via LawDispatcher) calls appropriate LawFunctor based on combination of IGeom and IPhys of the interaction. Again, it is an error if no functor capable of handling it is found.
6. LawDispatcher 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.

4.5.6 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 one to construct Body objects from functions such as utils.sphere only by knowing the requires Material type, enforcing the expectation of the model implementor.

4.6 Runtime structure

4.6.1 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.

4.6.2 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 acessible from anywhere in the code, without the necessity of passing pointers to such objects everywhere. The instance is created at startup and can be always retrieved (as non-const reference) using the `instance()` static method (e.g. `Omega::instance().getScene()`).

There are 3 singletons:

`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++
Omega.iter	`Scene::iter`
Omega.dt	`Scene::dt`
Omega.time	`Scene::time`
Omega.realtime	`Omega::getRealTime()`
Omega.stopAtIter	`Scene::stopAtIter`

`Omega` in c++ contains pointer to the current scene (`Omega::scene`, retrieved by `Omega::instance().getScene()`). Using Omega.switchScene, it is possible to swap this pointer with `Omega::sceneAnother`, a completely independent simulation. This can be useful for example (and this motivated this functionality) if while constructing simulation, another simulation has to be run to dynamically generate (i.e. by running simulation) packing of spheres.

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

4.6.3 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:

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().

4.7 Python framework

4.7.1 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:

and

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).

4.7.2 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* 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.

4.7.3 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.

4.7.4 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 "converters registry" is common for the whole program instance: there is no need to register converters in each script (by importing `_customConverters`, for instance), as that is done by yade at startup already.

Note: Custom converters only work for value that are passed by value to python (not "by reference"): some attributes defined using *YADE_CLASS_BASE_DOC_* macro family* are passed by value, but if you define your own, make sure that you read and understand Why is my automatic to-python conversion not being found?.

In short, the default for `def_readwrite` and `def_readonly` is to return references to underlying c++ objects, which avoids performing conversion on them. For that reason, return value policy must be set to `return_by_value` explicitly, using slighly more complicated `add_property` syntax, as explained at the page referenced.

[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.

4.8 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.

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

This has the following effects:

1. If file or directory has basename `OldClassName` (plus extension), it will be renamed using `bzr`.
2. All occurences of whole word `OldClassName` will be replaced by `NewClassName` in c++ sources.
3. An extry is added to py/system.py, which contains map of deprecated class names. At yade startup, proxy class with `OldClassName` will be created, which issues a `DeprecationWarning` when being instantiated, informing you of the new name you should use; it creates an instance of `NewClassName`, hence not disruting your script's functioning:

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

4.8.2 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.

4.9 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 priporuty than lower versions/revisions.

4.9.1 Prepare source package

Debian packaging files are located in debian/ directory. They contain build recipe debian/rules, dependecy 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).

4.9.2 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 5

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.

5.1 Packages

Packages are (as of now) provided for several Ubuntu versions from `Yade package archive` for stable and snapshot releases. 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:

- For stable releases:

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

- For latest builds from trunk:

  ```
  sudo add-apt-repository ppa:yade-pkg/snapshots    # for latest releases
  sudo apt-get update
  sudo apt-get install yade-bzr...
  ```

More detailed instructions are available at the corresponding pages of ppa's (links above).

5.2 Source code

Installation from source code is reasonable, when you want to add or modify constitutive laws, engines or functions... Installing the latest trunk version allows one to use newly added features, which are not yet available in packaged versions.

5.2.1 Download

If you want to install from source, you can install either a release (numbered version, which is frozen) or the current developement 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 that 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. developement 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, you can download any revision of the repository as compressed archive.

Release and trunk sources are compiled in the same way.

5.2.2 Prerequisities

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. The following commands have to be executed in command line of corresponding distributions. Just copy&paste to the terminal. To perform commands you should have root privileges

- **Ubuntu**, **Debian** and their derivatives:

  ```
  sudo apt-get install scons freeglut3-dev libloki-dev \
  libboost-date-time-dev libboost-filesystem-dev libboost-thread-dev \
  libboost-program-options-dev \
  libboost-regex-dev fakeroot dpkg-dev build-essential g++ \
  libboost-iostreams-dev python-dev libboost-python-dev ipython \
  python-matplotlib libsqlite3-dev python-numpy python-tk gnuplot doxygen \
  libgts-dev python-pygraphviz libvtk5-dev python-scientific bzr bzrtools libeigen2-dev \
  binutils-gold python-xlib python-qt4 pyqt4-dev-tools \
  gtk2-engines-pixbuf \
  libqglviewer-qt4-dev python-imaging libjs-jquery python-sphinx python-git python-bibtex
  ```

- **Fedora**:

  ```
  yum install scons qt3-devel freeglut-devel boost-devel boost-date-time \
  boost-filesystem boost-thread boost-regex fakeroot gcc gcc-c++ boost-iostreams \
  python-devel boost-python ipython python-matplotlib \
  sqlite-devel python-numeric ScientificPython-tk gnuplot doxygen gts-devel \
  ```

```
graphviz-python vtk-devel ScientificPython bzr bzrtools eigen2-devel libQGLViewer-devel \
loki-lib-devel python-xlib PyQt4 PyQt4-devel python-imaging python-sphinx python-bibtex
```

- **Gentoo (not good tested yet)**
  - Command to install all main dependencies:

    ```
    emerge dev-util/scons media-libs/freeglut media-libs/gd sys-libs/lib-compat-loki \
    dev-util/boost-build dev-libs/boost sys-apps/fakeroot app-arch/dpkg \
    sys-devel/gcc dev-libs/log4cxx dev-lang/python dev-python/ipython \
    dev-python/matplotlib dev-db/sqlite dev-python/numpy dev-lang/tk \
    sci-visualization/gnuplot app-doc/doxygen sci-libs/gts dev-python/pygraphviz \
    sci-libs/vtk dev-python/scientificpython dev-vcs/bzr dev-vcs/bzrtools dev-cpp/eigen \
    sys-devel/binutils dev-python/python-xlib x11-libs/qt-gui dev-python/PyQt4 dev-python/imaging \
    dev-python/sphinx dev-python/imaging dev-python/python-bibtex
    ```

  - `libqglviewer` is not in official Gentoo repository yet. But it can be installed from Gentoo Portage Overlay:

    ```
    emerge -va layman; emerge subversion; layman -f -a sunrise;\
    echo "source /usr/portage/local/layman/make.conf" >> /etc/make.conf \
    layman -s sunrise; layman -S; emerge x11-libs/libqglviewer
    ```

  - Probably `media-libs/gd` will require the following commands:

    ```
    mkdir /etc/portage -p;\
    echo "media-libs/gd fontconfig jpeg png truetype" >>/etc/portage/package.use
    ```

  - Some additional commands, which can be useful:

    ```
    ln -s /usr/lib/python2.6/site-packages/numpy/core/include/numpy/ /usr/include/
    ```

5.2.3 Compilation

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

```
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
```

Chapter 6

DEM Background

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

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

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

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

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

6.1 Collision detection

6.1.1 Generalities

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

1. fast collision detection using approximate predicate $\tilde{P}_i$ and $\tilde{P}_j$; they are pre-constructed in such a way as to abstract away individual features of P_i and P_j and satisfy the condition

$$\forall \mathbf{x} \in R^3 : x \in P_i \Rightarrow x \in \tilde{P}_i \tag{6.1}$$

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

$$\left(\tilde{P}_i \cap \tilde{P}_j\right) = \emptyset \Rightarrow \left(P_i \cap P_j\right) = \emptyset \tag{6.2}$$

   which is a candidate exclusion rule in the proper sense.

2. By filtering away impossible collisions in (6.2), a more expensive, exact collision detection algorithms can be run on possible interactions, filtering out remaining spurious couples $(\tilde{P}_i \cap \tilde{P}_j) \neq \emptyset \wedge \left(P_i \cap P_j\right) = \emptyset$. These algorithms operate on P_i and P_j and have to be able to handle all possible combinations of shape types.

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

6.1.2 Algorithms

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

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

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

> **Sweep and prune** algorithm operates on axis-aligned bounding boxes, which overlap if and only if they overlap along all axes. These algorithms have roughly $\mathcal{O}(n \log n)$ complexity, where n is number of particles as long as they exploit temporal coherence of the simulation.
>
> **Grid algorithms** represent continuous R^3 space by a finite set of regularly spaced points, leading to very fast neighbor search; they can reach the $\mathcal{O}(n)$ complexity [Munjiza1998] and recent research suggests ways to overcome one of the major drawbacks of this method, which is the necessity to adjust grid cell size to the largest particle in the simulation ([Munjiza2006], the "multistep" extension).

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

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

On a finer level, it is common to enlarge $\tilde{P}_i$ predicates in such a way that they satisfy the (6.1) condition during *several* timesteps; the first collision detection pass might then be run with stride, speeding up the simulation considerably. The original publication of this optimization by Verlet [Verlet1967] used enlarged list of neighbors, giving this technique the name *Verlet list*. In general cases, however, where neighbor lists are not necessarily used, the term *Verlet distance* is employed.

6.1.3 Sweep and prune

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

Presence of overlap of two Aabb's can be determined from conjunction of separate overlaps of intervals along each axis (fig-sweep-and-prune):

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

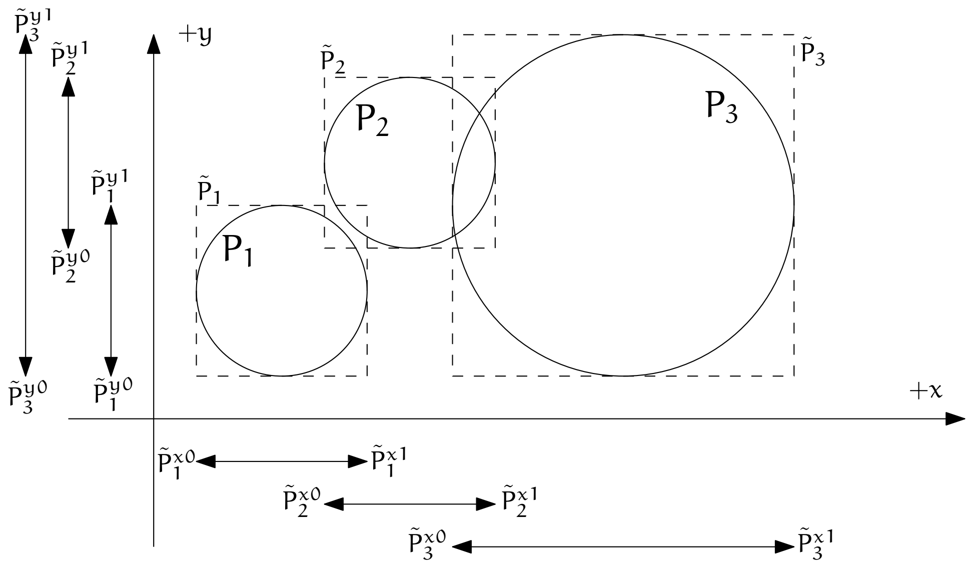


Figure 6.1: Sweep and prune algorithm (shown in 2D), where Aabb of each sphere is represented by minimum and maximum value along each axis. Spatial overlap of Aabb's is present if they overlap along all axes. In this case, $\tilde{P}_1 \cap \tilde{P}_2 \neq \emptyset$ (but note that $P_1 \cap P_2 = \emptyset$) and $\tilde{P}_2 \cap \tilde{P}_3 \neq \emptyset$.}

where (a, b) denotes interval in $\mathbb{R}$.

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

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

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

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

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

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

Aperiodic insertion sort

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

$$\| \quad 3 \quad 7 \quad 2 \quad 4 \quad \|$$

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

$(\leq)$	the sorting order with the left neighbor is correct, or
$(\|)$	the element is at the beginning of the sequence.

We start at the leftmost element (the current element is marked $\boxed{i}$)

$$\| \quad \boxed{3} \quad 7 \quad 2 \quad 4 \quad \|.$$

It obviously immediately satisfies $(\|)$, and we move to the next element:

$$\begin{tikzcd}[column sep=small] \| & 3 & \boxed{7} \arrow[l, bend left, "\leq"] & 2 & 4 & \|. \end{tikzcd}$$

Condition $(\leq)$ holds, therefore we move to the right. The $\boxed{2}$ is not in order (violating $(\leq)$) and two inversions take place; after that, $(\|)$ holds:

$$\begin{tikzcd}[column sep=small, row sep=small] \| & 3 & 7 & \boxed{2} \arrow[l, bend left, "\not\leq"] & 4 & \|, \\ \| & 3 & \boxed{2} \arrow[l, bend left, "\not\leq"] & 7 & 4 & \|, \\ \| & \boxed{2} & 3 & 7 & 4 & \|. \end{tikzcd}$$

The last element $\boxed{4}$ first violates $(\leq)$, but satisfies it after one inversion

$$\begin{tikzcd}[column sep=small, row sep=small] \| & 2 & 3 & 7 & \boxed{4} \arrow[l, bend left, "\not\leq"] & \|, \\ \| & 2 & 3 & \boxed{4} \arrow[l, bend left, "\leq"] & 7 & \|. \end{tikzcd}$$

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

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

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

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

Optimization with Verlet distances

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

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

$$L_{mn} = |X^n - X^m| \tag{6.3}$$

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

$$L_{mn} > \Delta L. \tag{6.4}$$

InsertionSortCollider.targetInterv is used to adjust ΔL independently for each particle. Larger ΔL will be assigned to the fastest ones, so that all particles would ideally reach the edge of their bounds after this “target” number of iterations. Results of using Verlet distance depend highly on the nature of simulation and choice of InsertionSortCollider.targetInterv. Adjusting the sizes independently for each particle is especially efficient if some parts of a problem have high-speed particles will others are not moving. If it is not the case, no significant gain should be expected as compared to targetInterv=0 (assigning the same ΔL to all particles).

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

6.2 Creating interaction between particles

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

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

6.2.1 Stiffnesses

Basic DEM interaction defines two stiffnesses: normal stiffness K_N and shear (tangent) stiffness K_T. It is desirable that K_N be related to fictitious Young's modulus of the particles' material, while K_T is typically determined as a given fraction of computed K_N. The K_T/K_N ratio determines macroscopic Poisson's ratio of the arrangement, which can be shown by dimensional analysis: elastic continuum has two parameters (E and ν) and basic DEM model also has 2 parameters with the same dimensions K_N and K_T/K_N; macroscopic Poisson's ratio is therefore determined solely by K_T/K_N and macroscopic Young's modulus is then proportional to K_N and affected by K_T/K_N.

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

Normal stiffness

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

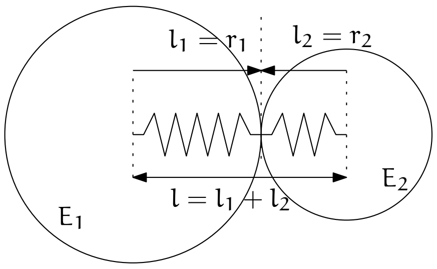


Figure 6.2: Series of 2 springs representing normal stiffness of contact between 2 spheres.

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

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

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

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

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

6.2.2 Other parameters

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

6.3 Strain evaluation

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

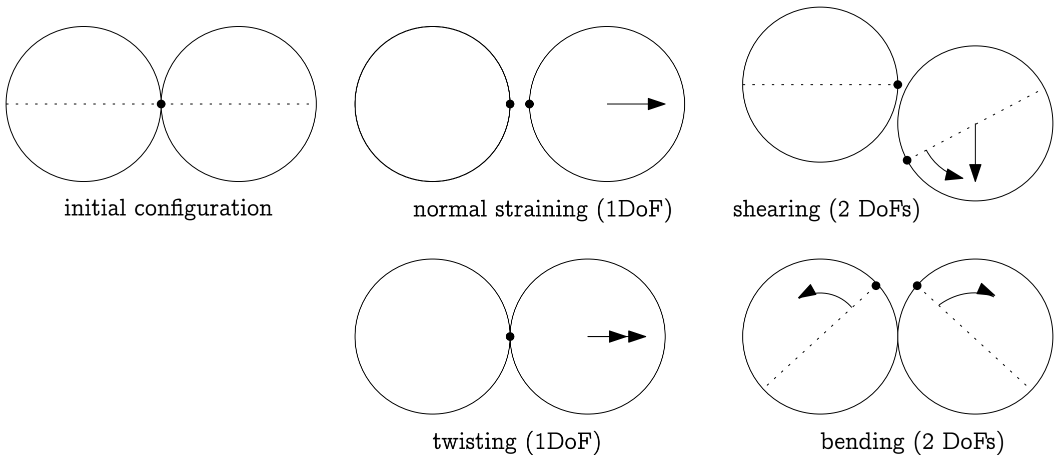


Figure 6.3: Degrees of freedom of configuration of two spheres. Normal strain appears if there is a difference of linear velocity along the interaction axis (n); shearing originates from the difference of linear velocities perpendicular to n *and* from the part of $\omega_1 + \omega_2$ perpendicular to n; twisting is caused by the part of $\omega_1 - \omega_2$ parallel with n; bending comes from the part of $\omega_1 - \omega_2$ perpendicular to n.

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

6.3.1 Normal strain

Constants

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

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

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

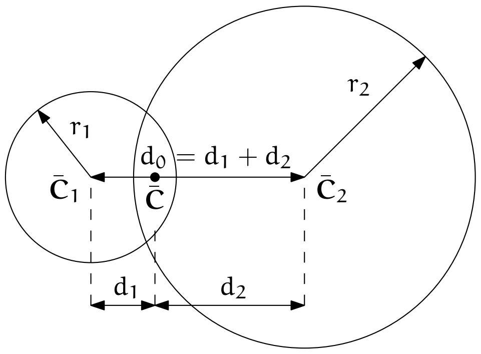


Figure 6.4: Geometry of the initial contact of 2 spheres; this case pictures spheres which already overlap when the contact is created (which can be the case at the beginning of a simulation) for the sake of generality. The initial contact point $\bar{C}$ is in the middle of the overlap zone.

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

$$d_0 \le R_I(r_1 + r_2). \tag{6.5}$$

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

Contact cross-section

Some constitutive laws are formulated with strains and stresses (Law2_Dem3DofGeom_CpmPhys_-Cpm, the concrete model described later, for instance); in that case, equivalent cross-section of the contact must be introduced for the sake of dimensionality. The exact definition is rather arbitrary; the CPM model (Ip2_CpmMat_CpmMat_CpmPhys) uses the relation

$$A_{eq} = \pi \min(r_1, r_2)^2 \tag{6.6}$$

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

Variables

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

$$\mathbf{n}^\circ = \frac{\mathbf{C}_2^\circ - \mathbf{C}_1^\circ}{|\mathbf{C}_2^\circ - \mathbf{C}_1^\circ|} \equiv \widehat{\mathbf{C}_2^\circ - \mathbf{C}_1^\circ} \tag{6.7}$$

and

$$\mathbf{C}^\circ = \mathbf{C}_1^\circ + \left(d_1 - \frac{d_0 - |\mathbf{C}_2^\circ - \mathbf{C}_1^\circ|}{2}\right)\mathbf{n}. \tag{6.8}$$

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

Normal displacement and strain can be defined as

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

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

For massively compressive simulations, it might be beneficial to use the logarithmic strain, such that the strain tends to $-\infty$ (rather than -1) as centers of both spheres approach. Otherwise, repulsive force would remain finite and the spheres could penetrate through each other. Therefore, we can adjust the definition of normal strain as follows:

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

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

6.3.2 Shear strain

In order to keep $\mathbf{u}_T$ consistent (e.g. that $\mathbf{u}_T$ must be constant if two spheres retain mutually constant configuration but move arbitrarily in space), then either $\mathbf{u}_T$ must track spheres' spatial motion or must (somehow) rely on sphere-local data exclusively.

These two possibilities lead to two algorithms of computing shear strains. They should give the same results (disregarding numerical imprecision), but there is a trade-off between computational cost of the incremental method and robustness of the total one.

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

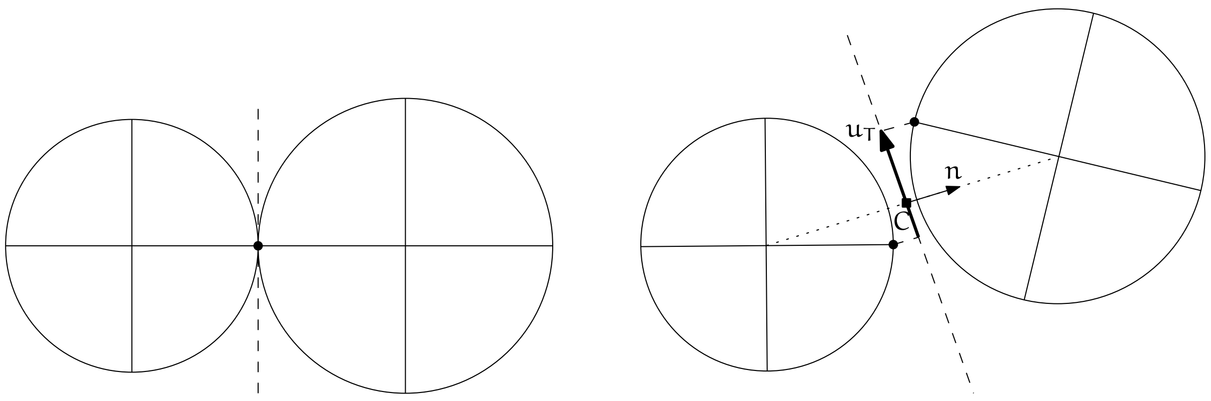


Figure 6.5: Evolution of shear displacement $\mathbf{u}_T$ due to mutual motion of spheres, both linear and rotational. Left configuration is the initial contact, right configuration is after displacement and rotation of one particle.

Incremental algorithm

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

1. Contact moves dues to changes of the spheres' positions $\mathbf{C}_1$ and $\mathbf{C}_2$, which updates current $\mathbf{C}^\circ$ and $\mathbf{n}^\circ$ as per (6.8) and (6.7). $\mathbf{u}_T^-$ is perpendicular to the contact plane at the previous step $\mathbf{n}^-$ and must be updated so that $\mathbf{u}_T^- + (\Delta\mathbf{u}_T) = \mathbf{u}_T^\circ \perp \mathbf{n}^\circ$; this is done by perpendicular projection to the plane first (which might decrease $|\mathbf{u}_T|$) and adding what corresponds to spatial rotation of the interaction instead:

$$(\Delta\mathbf{u}_T)_1 = -\mathbf{u}_T^- \times (\mathbf{n}^- \times \mathbf{n}^\circ)$$

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

2. Mutual movement of spheres, using only its part perpendicular to $\mathbf{n}^{\circ}$; $\mathbf{v}_{12}$ denotes mutual velocity of spheres at the contact point:

$$\begin{aligned}\mathbf{v}_{12} &= \left(\mathbf{v}_2^{\ominus} + \boldsymbol{\omega}_2^{-} \times (-d_2\mathbf{n}^{\circ})\right) - \left(\mathbf{v}_1^{\ominus} + \boldsymbol{\omega}_1^{\ominus} \times (d_1\mathbf{n}^{\circ})\right)\\ \mathbf{v}_{12}^{\perp} &= \mathbf{v}_{12} - (\mathbf{n}^{\circ}\cdot\mathbf{v}_{12})\mathbf{n}^{\circ}\\ (\Delta\mathbf{u}_T)_3 &= -\Delta t\mathbf{v}_{12}^{\perp}\end{aligned}$$

Finally, we compute

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

Total algorithm

The following algorithm, aiming at stabilization of response even with large rotation speeds or Δt approaching stability limit, was designed in [Smilauer2010b]. (A similar algorithm based on total formulation, which covers additionally bending and torsion, was proposed in [Wang2009].) It is based on tracking original contact points (with zero shear) in the particle-local frame.

In this section, variable symbols implicitly denote their current values unless explicitly stated otherwise.

Shear strain may have two sources: mutual rotation of spheres or transversal displacement of one sphere with respect to the other. Shear strain does not change if both spheres move or rotate but are not in linear or angular motion mutually. To accurately and reliably model this situation, for every new contact the initial contact point $\bar{\mathbf{C}}$ is mapped into local sphere coordinates ($\mathbf{p}_{01}$, $\mathbf{p}_{02}$). As we want to determine the distance between both points (i.e. how long the trajectory in on both spheres' surfaces together), the shortest path from current $\mathbf{C}$ to the initial locally mapped point on the sphere's surface is „unrolled“ to the contact plane ($\mathbf{p}'_{01}$, $\mathbf{p}'_{02}$); then we can measure their linear distance $\mathbf{u}_T$ and define shear strain $\boldsymbol{\varepsilon}_T = \mathbf{u}_T/d_0$ (fig. fig-shear-displacement).

More formally, taking $\bar{\mathbf{C}}_i$, $\bar{q}_i$ for the sphere initial positions and orientations (as quaterions) in global coordinates, the initial sphere-local contact point *orientation* (relative to sphere-local axis $\hat{x}$) is remembered:

$$\begin{aligned}\bar{\mathbf{n}} &= \widehat{\mathbf{C}_1 - \mathbf{C}_2},\\ \bar{q}_{01} &= \operatorname{Align}(\hat{x}, \bar{q}_1^{*}\bar{\mathbf{n}}\bar{q}_1^{**}),\\ \bar{q}_{02} &= \operatorname{Align}(\hat{x}, \bar{q}_2^{*}(-\bar{\mathbf{n}})\bar{q}_2^{**}).\end{aligned}$$

After some spheres motion, the original point can be “unrolled” to the current contact plane:

$$\begin{aligned}q &= \operatorname{Align}(\mathbf{n}, q_1\bar{q}_{01}\hat{x}(q_1\bar{q}_{01})^{*}) \quad \text{(auxiliary)}\\ \mathbf{p}'_{01} &= q_{\vartheta}d_1(q_{\mathbf{u}}\times\mathbf{n})\end{aligned}$$

where $q_{\mathbf{u}}$, q_{ϑ} are axis and angle components of q and p'_{01} is the unrolled point. Similarly,

$$\begin{aligned}q &= \operatorname{Align}(\mathbf{n}, q_2\bar{q}_{02}\hat{x}(q_2\bar{q}_{02})^{*})\\ \mathbf{p}'_{02} &= q_{\vartheta}d_1(q_{\mathbf{u}}\times(-\mathbf{n})).\end{aligned}$$

Shear displacement and strain are then computed easily:

$$\begin{aligned}\mathbf{u}_T &= \mathbf{p}'_{02} - \mathbf{p}'_{01}\\ \boldsymbol{\varepsilon}_T &= \frac{\mathbf{u}_T}{d_0}\end{aligned}$$

When using material law with plasticity in shear, it may be necessary to limit maximum shear strain, in which case the mapped points are moved closer together to the requested distance (without changing $\hat{\mathbf{u}}_T$). This allows us to remember the previous strain direction and also avoids summation of increments of plastic strain at every step (fig-shear-slip).

This algorithm is straightforwardly modified to facet-sphere interactions. In Yade, it is implemented by Dem3DofGeom and related classes.

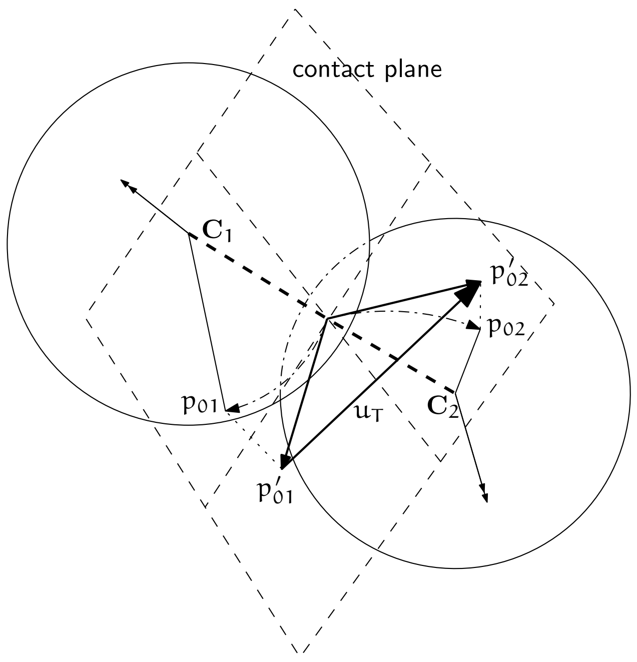


Figure 6.6: Shear displacement computation for two spheres in relative motion.

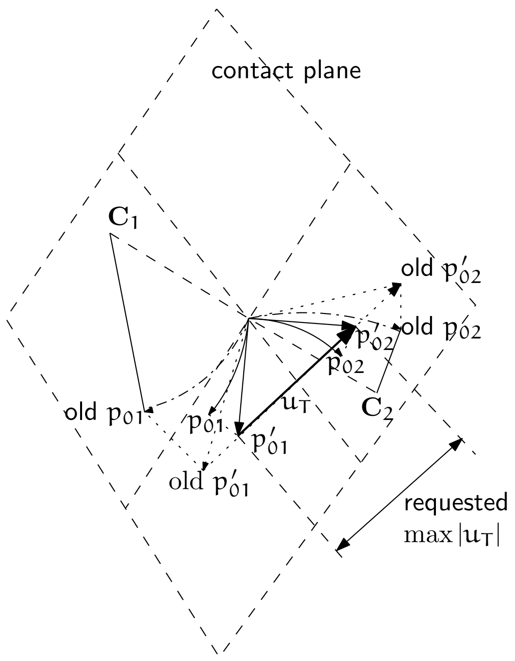


Figure 6.7: Shear plastic slip for two spheres.

6.4 Stress evaluation (example)

Once strain on a contact is computed, it can be used to compute stresses/forces acting on both spheres.

The constitutive law presented here is the most usual DEM formulation, originally proposed by Cundall. While the strain evaluation will be similar to algorithms described in the previous section regardless of stress evaluation, stress evaluation itself depends on the nature of the material being modeled. The constitutive law presented here is the most simple non-cohesive elastic case with dry friction, which Yade implements in Law2_Dem3DofGeom_FrictPhys_Basic (all constitutive laws derive from base class LawFunctor).

In DEM generally, some constitutive laws are expressed using strains and stresses while others prefer displacement/force formulation. The law described here falls in the latter category.

When new contact is established (discussed in *Engines*) it has its properties (IPhys) computed from Materials associated with both particles. In the simple case of frictional material FrictMat, Ip2_-FrictMat_FrictMat_FrictPhys creates a new FrictPhys instance, which defines normal stiffness K_N, shear stiffness K_T and friction angle φ.

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

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

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

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

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

$$\begin{aligned}\mathbf{F}_1 &+= \mathbf{F} & \mathbf{F}_2 &+= -\mathbf{F}\\ \mathbf{T}_1 &+= d_1(-\mathbf{n}) \times \mathbf{F} & \mathbf{T}_2 &+= d_2 \mathbf{n} \times \mathbf{F}.\end{aligned}$$

6.5 Motion integration

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

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

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

6.5.1 Position

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

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

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

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

from which we express

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

Typically, u^{-} is already not known (only u° is); we notice, however, that

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

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

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

which is (†); we arrive finally at

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

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

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

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

6.5.2 Orientation (spherical)

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

$$\dot{\omega}^{\circ}_i = T_i / I_{11},$$

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

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

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

$$\begin{aligned} (\Delta q)_{\vartheta} &= |\omega^{\oplus}|, \\ (\Delta q)_{u} &= \widehat{\omega^{\oplus}} \end{aligned}$$

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

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

6.5.3 Orientation (aspherical)

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

$$T_i = I_{ii}\dot{\omega}_i + (I_{kk} - I_{jj})\omega_j\omega_k.$$

Due to the presence of the current values of both $\boldsymbol{\omega}$ and $\dot{\boldsymbol{\omega}}$, they cannot be solved using the standard leapfrog algorithm (that was the case for translational motion and also for the spherical bodies' rotation where this equation reduced to $\mathbf{T} = \mathbf{I}\dot{\boldsymbol{\omega}}$).

The algorithm presented here is described by [Allen1989] (pg. 84–89) and was designed by Fincham for molecular dynamics problems; it is based on extending the leapfrog algorithm by mid-step/on-step estimators of quantities known at on-step/mid-step points in the basic formulation. Although it has received criticism and more precise algorithms are known ([Omelyan1999], [Neto2006], [Johnson2008]), this one is currently implemented in Yade for its relative simplicity.

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

For given particle in question, we know

- $\widetilde{\mathbf{I}}^\circ$ (constant) inertia matrix; diagonal, since in local, principal coordinates,
- $\mathbf{T}^\circ$ external torque,
- q° current orientation (and its equivalent rotation matrix $\mathbf{A}$),
- $\boldsymbol{\omega}^\ominus$ mid-step angular velocity,
- $\mathbf{L}^\ominus$ mid-step angular momentum; this is an auxiliary variable that must be tracked in addition for use in this algorithm. It will be zero in the initial step.

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

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

Then we compute $\dot{q}^\circ$, using q° and $\widetilde{\boldsymbol{\omega}}^\circ$:

$$\begin{aligned} \begin{pmatrix} \dot{q}^\circ_w \\ \dot{q}^\circ_x \\ \dot{q}^\circ_y \\ \dot{q}^\circ_z \end{pmatrix} &= \frac{1}{2}\begin{pmatrix} q^\circ_w & -q^\circ_x & -q^\circ_y & -q^\circ_z \\ q^\circ_x & q^\circ_w & -q^\circ_z & q^\circ_y \\ q^\circ_y & q^\circ_z & q^\circ_w & -q^\circ_x \\ q^\circ_z & -q^\circ_y & q^\circ_x & q^\circ_w \end{pmatrix}\begin{pmatrix} 0 \\ \widetilde{\omega}^\circ_x \\ \widetilde{\omega}^\circ_y \\ \widetilde{\omega}^\circ_z \end{pmatrix}, \\ q^\oplus &= q^\circ + \dot{q}^\circ\frac{\Delta t}{2}. \end{aligned}$$

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

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

6.5.4 Clumps (rigid aggregates)

DEM simulations frequently make use of rigid aggregates of particles to model complex shapes [Price2007] called *clumps*, typically composed of many spheres. Dynamic properties of clumps are computed from the properties of its members: the clump's mass m_c is summed over members, the inertia tensor $\mathbf{I}_c$ with respect to the clump's centroid is computed using the parallel axes theorem; local axes are oriented such that they are principal and inertia tensor is diagonal and clump's orientation is changed to compensate rotation of the local system, as to not change the clump members' positions in global space. Initial positions and orientations of all clump members in local coordinate system are stored.

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

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

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

6.5.5 Numerical damping

In simulations of quasi-static phenomena, it it desirable to dissipate kinetic energy of particles. Since most constitutive laws (including Law_ScGeom_FrictPhys_Basic shown above, *sect-formulation-stress-cundall*) do not include velocity-based damping (such as one in [Addetta2001]), it is possible to use artificial numerical damping. The formulation is described in [Pfc3dManual30], although our version is slightly adapted. The basic idea is to decrease forces which increase the particle velocities and vice versa by $(\Delta F)_d$, comparing the current acceleration sense and particle velocity sense. This is done by component, which makes the damping scheme clearly non-physical, as it is not invariant with respect to coordinate system rotation; on the other hand, it is very easy to compute. Cundall proposed the form (we omit particle indices i since it applies to all of them separately):

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

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

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

In Yade, we use the adapted form

$$\frac{(\Delta \mathbf{F})_{dw}}{\mathbf{F}_w} = -\lambda_d \operatorname{sgn} \mathbf{F}_w \underbrace{\left(\dot{u}^{\ominus}_w + \frac{\ddot{u}^{\circ}_w \Delta t}{2}\right)}_{\simeq \dot{u}^{\circ}_w}, \tag{6.9}$$

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

In Yade, damping (6.9) is implemented in the NewtonIntegrator engine; the damping coefficient λ_d is NewtonIntegrator.damping.

6.5.6 Stability considerations

Critical timestep

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

$$\Delta t_{cr} = \frac{2}{\omega_{max}} \tag{6.10}$$

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

Single mass-spring system

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

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

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

$$\omega^{(1)} = \sqrt{\frac{K}{m}} \tag{6.11}$$

does not depend on initial conditions. Since there is one single mass, $\omega^{(1)}_{max} = \omega^{(1)}$. Plugging (6.11) into (6.10), we obtain

$$\Delta t^{(1)}_{cr} = 2/\omega^{(1)}_{max} = 2\sqrt{m/K}$$

for a single oscillator.

General mass-spring system

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

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

The overall critical timestep is then

$$\Delta t_{cr} = \frac{2}{\omega_{max}} = \min_i 2\sqrt{\frac{m_i}{K^{(2)}_i}} = \min_i 2\sqrt{\frac{m_i}{2K_i}} = \min_i \sqrt{2}\sqrt{\frac{m_i}{K_i}}. \tag{6.12}$$

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

$$\Delta t_{cr} = \min \Delta t_{cr\,k}, \quad k \in \{1, ..., 6\}. \tag{6.13}$$

DEM simulations

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

$$\mathbf{K}_{ij} = \sum_k (K_{Nk} - K_{Tk})\mathbf{n}_i\mathbf{n}_j + K_{Tk} = \sum_j K_{Nk}\left((1-\xi)\mathbf{n}_i\mathbf{n}_j + \xi\right) \tag{6.14}$$

with i and $j \in \{x, y, z\}$. Equations (6.13) and (6.14) determine Δt_{cr} in a simulation. A similar approach generalized to all 6 DOFs is implemented by the GlobalStiffnessTimeStepper engine in Yade. The derivation of generalized stiffness including rotational terms is very similar but not developped here, for simplicity. For full reference, see “PFC3D - Theoretical Background”.

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

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

Estimation of Δt_{cr} by wave propagation speed

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

In Yade, particles have associated Material which defines density ρ (Material.density), and also may define (in ElastMat and derived classes) particle’s “Young’s modulus” E (ElastMat.young). ρ is used when particle’s mass m is initially computed from its ρ, while E is taken in account when creating new interaction between particles, affecting stiffness K_N. Knowing m and K_N, we can estimate (6.14) for each particle; we obviously neglect

- number of interactions per particle N_i; for a “reasonable” radius distribution, however, there is a geometrically imposed upper limit (6 for a packing of spheres with equal radii, for instance);
- the exact relationship the between particles’ rigidities E_i, E_j, supposing only that K_N is somehow proportional to them.

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

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

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

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

This algorithm is implemented in the utils.PWaveTimeStep function.

Let us compare this result to (6.12); this necessitates making several simplifying hypotheses:

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

  $$K_N = E\pi' R', \tag{6.15}$$

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

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

We try to express the average per-particle stiffness from (6.14). It is a sum over all interactions where K_N and ξ are scalars that will not rotate with interaction, while n_w is w-th component of unit interaction normal n. Since we supposed uniform spatial distribution, we can replace n_w^2 by its average value $\overline{n}_w^2$. Recognizing components of n as direction cosines, the average values of n_w^2 is $1/3$. %we find the average value by integrating over all possible orientations, which are uniformly distributed in space:

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

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

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

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

we substitute K into (6.12) using (6.15):

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

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

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

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

Concrete particle model computes contact stiffness from the equivalent area A_{eq} first (6.6),

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

d_0 is the initial contact length, which will be, for interaction radius (6.5) $R_I > 1$, in average larger than $2R$. For $R_I = 1.5$ (sect.~ref{sect-calibration-elastic-properties}), we can roughly estimate $\overline{d}_0 = 1.25 \cdot 2R = \frac{5}{2}R$, getting

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

where $\frac{2}{5}\pi = \pi'$ by comparison with (6.15).

Interaction radius $R_I = 1.5$ leads to average $N \approx 12$ interactions per sphere for dense packing of spheres with the same radius R. $\xi = 0.2$ is calibrated (sect.~ref{sect-calibration-elastic-properties}) to match the desired macroscopic Poisson's ratio $\nu = 0.2$.

Finally, we obtain the ratio

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

showing significant overestimation by the p-wave algorithm.

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

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

which again overestimates the numerical critical timestep.

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

Non-elastic Δt constraints

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

6.6 Periodic boundary conditions

While most DEM simulations happen in R^3 space, it is frequently useful to avoid boundary effects by using periodic space instead. In order to satisfy periodicity conditions, periodic space is created by repetition of parallelepiped-shaped cell. In Yade, periodic space is implemented in the Cell class. The geometry of the cell in the reference coordinates system is defined by three edges of the parallepiped. The corresponding base vectors are stored in the columns of matrix $\mathbf{H}$ (Cell.hSize).

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

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

6.6.1 Deformations handling

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

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

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

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

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

6.6.2 Collision detection in periodic cell

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

Approximate collision detection

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

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

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

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

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

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

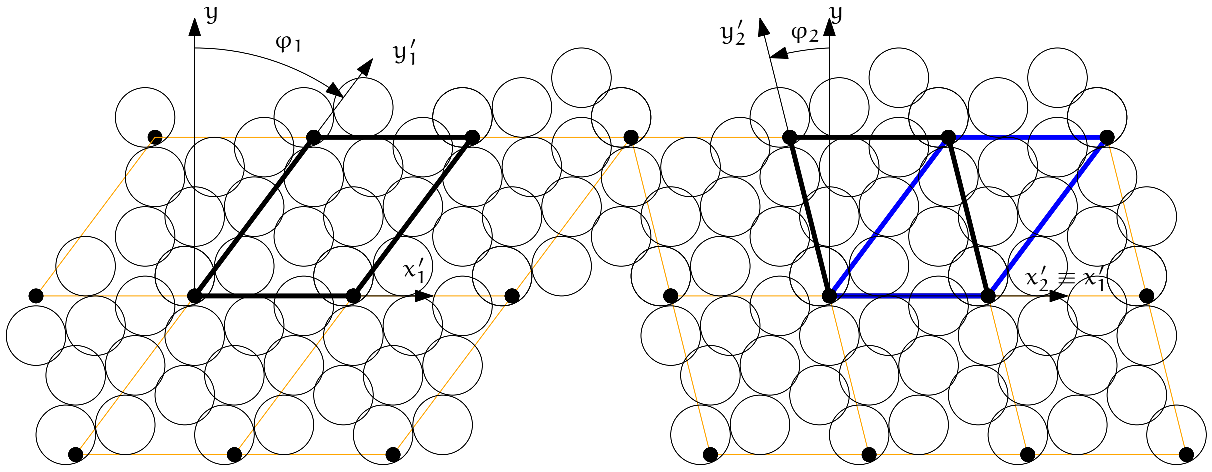


Figure 6.8: Flipping cell (utils.flipCell) to avoid infinite stretch of the bounding boxes' spans with growing φ. Cell flip does not affect interactions from the point of view of the simulation. The periodic arrangement on the left is the same as the one on the right, only the cell is situated differently between identical grid points of repetition; at the same time $|\varphi_2| < |\varphi_1|$ and sphere bounding box's x-span stretched by $1/\cos\varphi$ becomes smaller. Flipping can be repeated, making effective infinite shear possible.

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

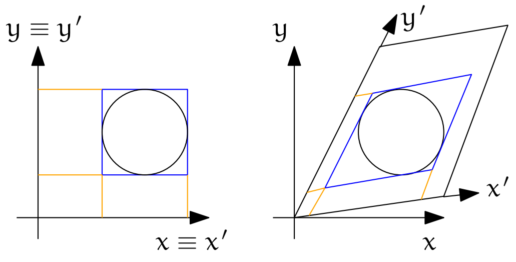


Figure 6.9: Constructing axis-aligned bounding box (Aabb) of a sphere in simulation space coordinates (without periodic cell – left) and transformed cell coordinates (right), where collision detection axes x', y' are not identical with simulation space axes x, y. Bounds' projection to axes is shown by orange lines.

Exact collision detection

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

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

Periodic insertion sort algorithm

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

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

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

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

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

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

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

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

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

$$\begin{tikzcd}
4_1 & 12_2 & \| & \boxed{-1_2} & -2_4 & 5_0, \\
4_1 & 12_2 & \boxed{9_1} \arrow[l, bend left, "\not\leq"] & \| & -2_4 & 5_0, \\
4_1 & \boxed{9_1} \arrow[l, bend left, "\leq"] & 12_2 & \| & -2_4 & 5_0.
\end{tikzcd}$$

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

$$\begin{tikzcd}
4_1 & 9_1 & 12_2 & \| & \boxed{-2_4} & 5_0, \\
4_1 & 9_1 & 12_2 & \boxed{8_3} \arrow[l, bend left, "\not\leq"] & \| & 5_0, \\
4_1 & 9_1 & \boxed{8_3} \arrow[l, bend left, "\not\leq"] & 12_2 & \| & 5_0, \\
4_1 & \boxed{8_3} \arrow[l, bend left, "\leq"] & 9_1 & 12_2 & \| & 5_0.
\end{tikzcd}$$

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

$$\begin{tikzcd}
4_1 & 8_3 & 9_1 & 12_2 & \| & \boxed{5_0} \arrow[ll, bend left, "\not\leq"], \\
4_1 & 8_3 & 9_1 & \boxed{5_{-1}} \arrow[l, bend left, "\not\leq"] & \| & 2_3, \\
4_1 & 8_3 & \boxed{5_{-1}} \arrow[l, bend left, "\not\leq"] & 9_1 & \| & 2_3, \\
4_1 & \boxed{5_{-1}} \arrow[l, bend left, "\leq"] & 8_3 & 9_1 & \| & 2_3.
\end{tikzcd}$$

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

$$\begin{tikzcd}
\boxed{4_1} \arrow[rrrrr, bend right, "\geq"] & 5_{-1} & 8_3 & 9_1 & \| & 2_3
\end{tikzcd}$$

and so is the last element

$$\begin{tikzcd}
4_1 & \boxed{5_{-1}} \arrow[l, bend left, "\leq"] & 8_3 & 9_1 & \| & 2_3.
\end{tikzcd}$$

6.7 Computational aspects

6.7.1 Cost

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

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

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

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

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

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

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

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

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

Let us suppose linear scaling. Additionally, let us suppose that the material to be simulated (E, ρ) and the simulation setup (V, T) are given in advance. Finally, dimensionless constants s_t, p and n_{cpu} will have a fixed value. This leaves us with one last degree of freedom, r. We may write

$$Z \propto iN \frac{1}{n_{\mathrm{cpu}}} = \frac{T}{s_t r}\sqrt{\frac{E}{\rho}} p \frac{V}{\frac{4}{3}\pi r^3} \frac{1}{n_{\mathrm{cpu}}} \propto \frac{1}{r}\frac{1}{r^3} = \frac{1}{r^4}.$$

This (rather trivial) result is essential to realize DEM scaling; if we want to have finer results, refining the “mesh” by halving r, the computation time will grow $2^4 = 16$ times.

For very crude estimates, one can use a known simulation to obtain a machine “constant”

$$\mu = \frac{Z}{Ni}$$

with the meaning of time per particle and per timestep (in the order of 10^{-6} s for current machines). μ will be only useful if simulation characteristics are similar and non-linearities in scaling do not have major influence, i.e. N should be in the same order of magnitude as in the reference case.

6.7.2 Result indeterminism

It is naturally expected that running the same simulation several times will give exactly the same results: although the computation is done with finite precision, round-off errors would be deterministically the same at every run. While this is true for *single-threaded* computation where exact order of all operations is given by the simulation itself, it is not true anymore in *multi-threaded* computation which is described in detail in later sections.

The straight-forward manner of parallel processing in explicit DEM is given by the possibility of treating interactions in arbitrary order. Strain and stress is evaluated for each interaction independently, but forces from interactions have to be summed up. If summation order is also arbitrary (in Yade, forces are accumulated for each thread in the order interactions are processed, then summed together), then the results can be slightly different. For instance

```
(1/10.)+(1/13.)+(1/17.)=0.23574660633484162
(1/17.)+(1/13.)+(1/10.)=0.23574660633484165
```

As forces generated by interactions are assigned to bodies in quasi-random order, summary force F_i on the body can be different between single-threaded and multi-threaded computations, but also between different runs of multi-threaded computation with exactly the same parameters. Exact thread scheduling by the kernel is not predictable since it depends on asynchronous events (hardware interrupts) and other unrelated tasks running on the system; and it is thread scheduling that ultimately determines summation order of force contributions from interactions.

Numerical damping influence

The effect of summation order can be significantly amplified by the usage of a *discontinuous* damping function in NewtonIntegrator given in (6.9) as

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

If the sgn argument is close to zero then the least significant finite precision artifact can determine whether the equation (relative increment of F_w) is $+\lambda_d$ or $-\lambda_d$. Given commonly used values of $\lambda_d = 0.4$, it means that such artifact propagates from least significant place to the most significant one at once.

Chapter 7

Class reference (yade.wrapper module)

7.1 Bodies

7.1.1 Body

class `yade.wrapper.Body`(*inherits Serializable*)
A particle, basic element of simulation; interacts with other bodies.

`aspherical`(*=false*)
Whether this body has different inertia along principal axes; NewtonIntegrator makes use of this flag to call rotation integration routine for aspherical bodies, which is more expensive.

`bound`(*=uninitalized*)
Bound, approximating volume for the purposes of collision detection.

`bounded`(*=true*)
Whether this body should have Body.bound created. Note that bodies without a bound do not participate in collision detection. (In c++, use `Body::isBounded`/`Body::setBounded`)

`clumpId`
Id of clump this body makes part of; invalid number if not part of clump; see Body::isStandalone, Body::isClump, Body::isClumpMember properties.

Not meant to be modified directly from Python, use O.bodies.appendClumped instead.

`dynamic`(*=true*)
Whether this body will be moved by forces. (In c++, use `Body::isDynamic`/`Body::setDynamic`)

`flags`(*=FLAG_BOUNDED*)
Bits of various body-related flags. *Do not access directly.* In c++, use isDynamic/setDynamic, isBounded/setBounded, isAspherical/setAspherical. In python, use Body.dynamic, Body.bounded, Body.aspherical.

`groupMask`(*=1*)
Bitmask for determining interactions.

`id`(*=Body::ID_NONE*)
Unique id of this body.

`intrs`() → list
Return all interactions in which this body participates.

isClump
True if this body is clump itself, false otherwise.

isClumpMember
True if this body is clump member, false otherwise.

isStandalone
True if this body is neither clump, nor clump member; false otherwise.

iterBorn
Returns step number at which the body was added to simulation.

mask
Shorthand for Body::groupMask

mat
Shorthand for Body::material

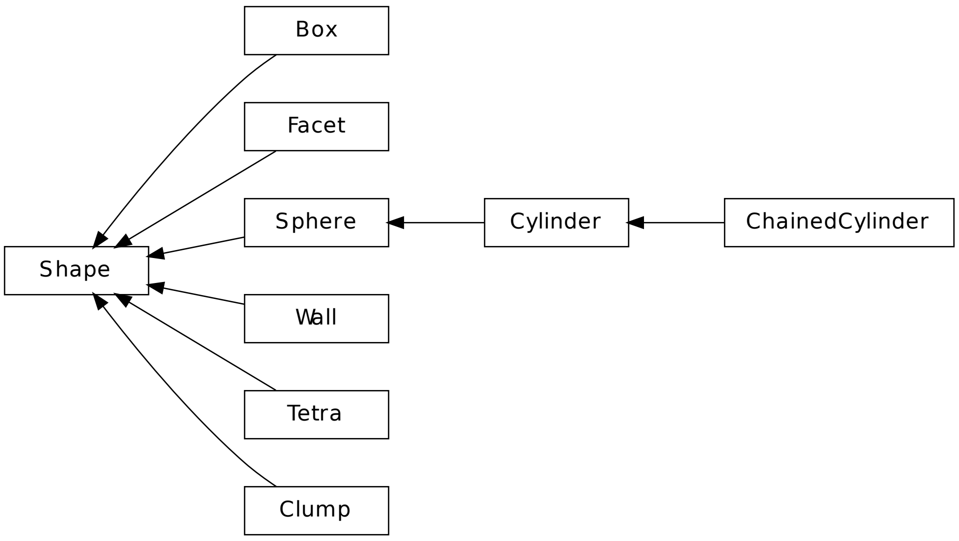
material(*=uninitalized*)
Material instance associated with this body.

shape(*=uninitalized*)
Geometrical Shape.

state(*=new State*)
Physical state.

timeBorn
Returns time at which the body was added to simulation.

7.1.2 Shape



class `yade.wrapper.`**Shape**(*inherits Serializable*)
Geometry of a body

color(*=Vector3r(1, 1, 1)*)
Color for rendering (normalized RGB).

dispHierarchy($\left[(bool)names{=}True\right]$) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

highlight(*=false*)
Whether this Shape will be highlighted when rendered.

wire(*=false*)
Whether this Shape is rendered using color surfaces, or only wireframe (can still be overridden by global config of the renderer).

class yade.wrapper.**Box**(*inherits Shape → Serializable*)
Box (cuboid) particle geometry. (Avoid using in new code, prefer Facet instead.

extents(*=uninitalized*)
Half-size of the cuboid

class yade.wrapper.**ChainedCylinder**(*inherits Cylinder → Sphere → Shape → Serializable*)
Geometry of a deformable chained cylinder, using geometry Cylinder.

chainedOrientation(*=Quaternionr::Identity()*)
Deviation of node1 orientation from node-to-node vector

initLength(*=0*)
tensile-free length, used as reference for tensile strain

class yade.wrapper.**Clump**(*inherits Shape → Serializable*)
Rigid aggregate of bodies

members
Return clump members as {'id1':(relPos,relOri),...}

class yade.wrapper.**Cylinder**(*inherits Sphere → Shape → Serializable*)
Geometry of a cylinder, as Minkowski sum of line and sphere.

length(*=NaN*)
Length [m]

segment(*=Vector3r::Zero()*)
Length vector

class yade.wrapper.**Facet**(*inherits Shape → Serializable*)
Facet (triangular particle) geometry.

normal(*=Vector3r(NaN, NaN, NaN)*)
Facet's normal

vertices(*=vector<Vector3r>(3, Vector3r(NaN, NaN, NaN))*)
Vertex positions in local coordinates.

class yade.wrapper.**Sphere**(*inherits Shape → Serializable*)
Geometry of spherical particle.

radius(*=NaN*)
Radius [m]

class yade.wrapper.**Tetra**(*inherits Shape → Serializable*)
Tetrahedron geometry.

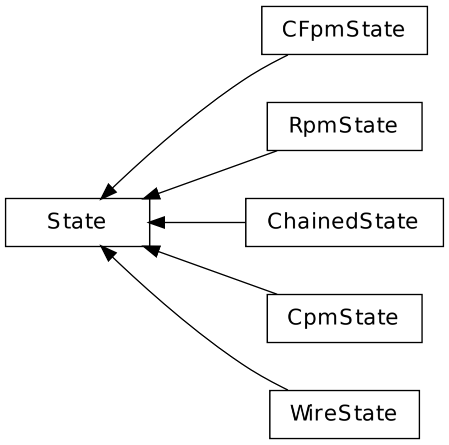
v(*=std::vector<Vector3r>(4)*)
Tetrahedron vertices in global coordinate system.

class yade.wrapper.**Wall**(*inherits Shape → Serializable*)
Object representing infinite plane aligned with the coordinate system (axis-aligned wall).

axis(*=0*)
Axis of the normal; can be 0,1,2 for +x, +y, +z respectively (Body's orientation is disregarded for walls)

sense(*=0*)
Which side of the wall interacts: -1 for negative only, 0 for both, +1 for positive only

7.1.3 State



class `yade.wrapper.`**`State`**(*inherits Serializable*)

State of a body (spatial configuration, internal variables).

`angMom`(*=Vector3r::Zero()*)
: Current angular momentum

`angVel`(*=Vector3r::Zero()*)
: Current angular velocity

`blockedDOFs`
: Degress of freedom where linear/angular velocity will be always constant (equal to zero, or to an user-defined value), regardless of applied force/torque. String that may contain 'xyzXYZ' (translations and rotations).

`dispHierarchy`($\left[\textit{(bool)names=True} \right]$) → list
: Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

`dispIndex`
: Return class index of this instance.

`displ`() → Vector3
: Displacement from reference position (pos - refPos

`inertia`(*=Vector3r::Zero()*)
: Inertia of associated body, in local coordinate system.

`isDamped`(*=true*)
: Damping in Newtonintegrator can be deactivated for individual particles by setting this variable to FALSE. E.g. damping is inappropriate for particles in free flight under gravity but it might still be applicable to other particles in the same simulation.

`mass`(*=0*)
: Mass of this body

`ori`
: Current orientation.

`pos`
: Current position.

`refOri`(*=Quaternionr::Identity()*)
: Reference orientation

refPos(*=Vector3r::Zero()*)
Reference position

rot() → Vector3
Rotation from reference orientation (as rotation vector)

se3(*=Se3r(Vector3r::Zero(), Quaternionr::Identity())*)
Position and orientation as one object.

vel(*=Vector3r::Zero()*)
Current linear velocity.

class yade.wrapper.**CFpmState**(*inherits State → Serializable*)
CFpm state information about each body.

None of that is used for computation (at least not now), only for post-processing.

numBrokenCohesive(*=0*)
Number of broken cohesive links. [-]

class yade.wrapper.**ChainedState**(*inherits State → Serializable*)
State of a chained bodies, containing information on connectivity in order to track contacts jumping over contiguous elements. Chains are 1D lists from which id of chained bodies are retrieved via :yref:rank<ChainedState::rank>‘ and :yref:chainNumber<ChainedState::chainNumber>‘.

addToChain(*(int)bodyId*) → None
Add body to current active chain

bId(*=-1*)
id of the body containing - for postLoad operations only

chainNumber(*=0*)
chain id

currentChain = 0

rank(*=0*)
rank in the chain

class yade.wrapper.**CpmState**(*inherits State → Serializable*)
State information about body use by *cpm-model*.

None of that is used for computation (at least not now), only for post-processing.

damageTensor(*=Matrix3r::Zero()*)
Damage tensor computed with microplane theory averaging. state.damageTensor.trace() = state.normDmg

epsPlBroken(*=0*)
Plastic strain on contacts already deleted (bogus values)

epsVolumetric(*=0*)
Volumetric strain around this body (unused for now)

normDmg(*=0*)
Average damage including already deleted contacts (it is really not damage, but 1-relResidualStrength now)

normEpsPl(*=0*)
Sum of plastic strains normalized by number of contacts (bogus values)

numBrokenCohesive(*=0*)
Number of (cohesive) contacts that damaged completely

numContacts(*=0*)
Number of contacts with this body

stress(*=Matrix3r::Zero()*)
Stress tensor of the spherical particle (under assumption that particle volume = pi*r*r*r*4/3.). To get actual stress, multiply this value by packing fraction (for random dense packing something like 0.63)

class yade.wrapper.**RpmState**(*inherits State → Serializable*)
State information about Rpm body.

specimenMass(*=0*)
Indicates the mass of the whole stone, which owns the particle.

specimenMaxDiam(*=0*)
Indicates the maximal diametr of the specimen.

specimenNumber(*=0*)
The variable is used for particle size distribution analyze. Indicates, to which part of specimen belongs para of particles.

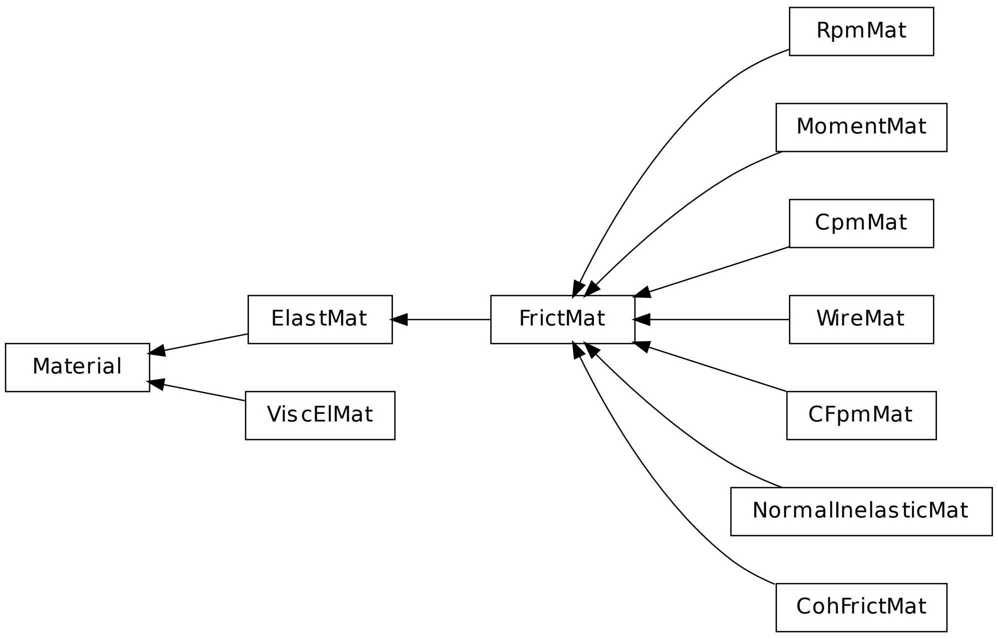
specimenVol(*=0*)
Indicates the mass of the whole stone, which owns the particle.

class yade.wrapper.**WireState**(*inherits State → Serializable*)
Wire state information of each body.

None of that is used for computation (at least not now), only for post-processing.

numBrokenLinks(*=0*)
Number of broken links (e.g. number of wires connected to the body which are broken). [-]

7.1.4 Material



class yade.wrapper.**Material**(*inherits Serializable*)
Material properties of a body.

density(*=1000*)
Density of the material [kg/m^3]

dispHierarchy($\left[\textit{(bool)names=True}\right]$) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

`id`(*=-1, not shared*)
: Numeric id of this material; is non-negative only if this Material is shared (i.e. in O.materials), -1 otherwise. This value is set automatically when the material is inserted to the simulation via O.materials.append. (This id was necessary since before boost::serialization was used, shared pointers were not tracked properly; it might disappear in the future)

`label`(*=uninitalized*)
: Textual identifier for this material; can be used for shared materials lookup in MaterialContainer.

`newAssocState()` → State
: Return new State instance, which is associated with this Material. Some materials have special requirement on Body::state type and calling this function when the body is created will ensure that they match. (This is done automatically if you use utils.sphere, … functions from python).

class `yade.wrapper.CFpmMat`(*inherits FrictMat → ElastMat → Material → Serializable*)
cohesive frictional material, for use with other CFpm classes

`type`(*=0*)
: Type of the particle. If particles of two different types interact, it will be with friction only (no cohesion).[-]

class `yade.wrapper.CohFrictMat`(*inherits FrictMat → ElastMat → Material → Serializable*)

`alphaKr`(*=2.0*)
: Dimensionless coefficient used for the rolling stiffness.

`alphaKtw`(*=2.0*)
: Dimensionless coefficient used for the twist stiffness.

`etaRoll`(*=-1.*)
: Dimensionless coefficient used to calculate the plastic rolling moment (if negative, plasticity will not be applied).

`isCohesive`(*=true*)

`momentRotationLaw`(*=false*)
: Use bending/twisting moment at contact. The contact will have moments only if both bodies have this flag true. See CohFrictPhys::cohesionDisablesFriction for details.

`normalCohesion`(*=0*)

`shearCohesion`(*=0*)

class `yade.wrapper.CpmMat`(*inherits FrictMat → ElastMat → Material → Serializable*)
Concrete material, for use with other Cpm classes.

The model is contained in externally defined macro CPM_MATERIAL_MODEL, which features damage in tension, plasticity in shear and compression and rate-dependence. For commercial reasons, rate-dependence and compression-plasticity is not present in reduced version of the model, used when CPM_MATERIAL_MODEL is not defined. The full model will be described in detail in my (Václav Šmilauer) thesis along with calibration procedures (rigidity, poisson's ratio, compressive/tensile strength ratio, fracture energy, behavior under confinement, rate-dependent behavior).

Even the public model is useful enough to run simulation on concrete samples, such as uniaxial tension-compression test.

`G_over_E`(*=NaN*)
: Ratio of normal/shear stiffness at interaction level [-]

`crackOpening`(*=NaN*)
: Crack opening when the crack is fully broken in tension. [m]

`damLaw`(*=1*)
: Law for gamage evolution in uniaxial tension. 0 for linear stress-strain softening branch, 1 for exponential damage evolution law

dmgRateExp(*=0*)
: Exponent for normal viscosity function. [-]

dmgTau(*=-1, deactivated if negative*)
: Characteristic time for normal viscosity. [s]

epsCrackOnset(*=NaN*)
: Limit elastic strain [-]

isoPrestress(*=0*)
: Isotropic prestress of the whole specimen. [Pa]

neverDamage(*=false*)
: If true, no damage will occur (for testing only).

plRateExp(*=0*)
: Exponent for visco-plasticity function. [-]

plTau(*=-1, deactivated if negative*)
: Characteristic time for visco-plasticity. [s]

relDuctility(*=NaN*)
: relative ductility of bonds in normal direction

sigmaT(*=NaN*)
: Initial cohesion [Pa]

class yade.wrapper.**ElastMat**(*inherits Material → Serializable*)
: Purely elastic material. The material parameters may have different meanings depending on the IPhysFunctor used : true Young and Poisson in Ip2_FrictMat_FrictMat_MindlinPhys, or contact stiffnesses in Ip2_FrictMat_FrictMat_FrictPhys.

poisson(*=.25*)
: Poisson's ratio or the ratio between shear and normal stiffness [-]. It has different meanings depending on the Ip functor.

young(*=1e9*)
: elastic modulus [Pa]. It has different meanings depending on the Ip functor.

class yade.wrapper.**FrictMat**(*inherits ElastMat → Material → Serializable*)
: Elastic material with contact friction. See also ElastMat.

frictionAngle(*=.5*)
: Contact friction angle (in radians). Hint : use 'radians(degreesValue)' in python scripts.

class yade.wrapper.**MomentMat**(*inherits FrictMat → ElastMat → Material → Serializable*)
: Material for constitutive law of (Plassiard & al., 2009); see Law2_SCG_MomentPhys_CohesionlessMomentRotation for details.

Users can input eta (constant for plastic moment) to Spheres and Boxes. For more complicated cases, users can modify TriaxialStressController to use different eta values during isotropic compaction.

eta(*=0*)
: (has to be stored in this class and not by ContactLaw, because users may want to change its values before/after isotropic compaction.)

class yade.wrapper.**NormalInelasticMat**(*inherits FrictMat → ElastMat → Material → Serializable*)
: Material class for particles whose contact obey to a normal inelasticity (governed by this coeff_dech).

coeff_dech(*=1.0*)
: =kn(unload) / kn(load)

class yade.wrapper.**RpmMat**(*inherits FrictMat → ElastMat → Material → Serializable*)
: Rock material, for use with other Rpm classes.

Brittleness(*=0*)
: One of destruction parameters. [-] //(Needs to be reworked)

G_over_E(*=1*)
Ratio of normal/shear stiffness at interaction level. [-]

exampleNumber(*=0*)
Number of the specimen. This value is equal for all particles of one specimen. [-]

initCohesive(*=false*)
The flag shows, whether particles of this material can be cohesive. [-]

stressCompressMax(*=0*)
Maximal strength for compression. The main destruction parameter. [Pa] //(Needs to be reworked)

class `yade.wrapper.`**ViscElMat**(*inherits Material → Serializable*)
Material for simple viscoelastic model of contact.

Note: `Shop::getViscoelasticFromSpheresInteraction` (and utils.getViscoelasticFromSpheresInteraction in python) compute kn, cn, ks, cs from analytical solution of a pair spheres interaction problem.

cn(*=NaN*)
Normal viscous constant

cs(*=NaN*)
Shear viscous constant

frictionAngle(*=NaN*)
Friction angle [rad]

kn(*=NaN*)
Normal elastic stiffness

ks(*=NaN*)
Shear elastic stiffness

class `yade.wrapper.`**WireMat**(*inherits FrictMat → ElastMat → Material → Serializable*)
Material for use with the Wire classes

as(*=0*)
Cross-section area of a single wire used for the computation of the limit normal contact forces. [m^2]

diameter(*=0.0027*)
Diameter of the single wire in [m] (the diameter is used to compute the cross-section area of the wire).

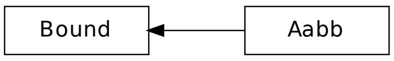
isDoubleTwist(*=false*)
Type of the mesh. If true two particles of the same material which body ids differ by one will be considered as double-twisted interaction.

lambdaEps(*=0.4*)
Parameter between 0 and 1 to reduce the failure strain of the double-twisted wire (as used by [Bertrand2008]). [-]

lambdak(*=0.21*)
Parameter between 0 and 1 to compute the elastic stiffness of the double-twisted wire (as used by [Bertrand2008]): $k^D = 2(\lambda_k k_h + (1-\lambda_k)k^S)$. [-]

strainStressValues(*=uninitalized*)
Piecewise linear definition of the stress-strain curve by set of points (strain[-]>0,stress[Pa]>0) for one single wire. Tension only is considered and the point (0,0) is not needed!

7.1.5 Bound



class `yade.wrapper.Bound`(*inherits Serializable*)
Object bounding part of space taken by associated body; might be larger, used to optimalize collision detection

`color`(*=Vector3r(1, 1, 1)*)
Color for rendering this object

`dispHierarchy`($\left[(bool)names=True\right]$) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

`dispIndex`
Return class index of this instance.

`lastUpdateIter`(*=0*)
record iteration of last reference position update *(auto-updated)*

`max`(*=Vector3r(NaN, NaN, NaN)*)
Lower corner of box containing this bound (and the Body as well)

`min`(*=Vector3r(NaN, NaN, NaN)*)
Lower corner of box containing this bound (and the Body as well)

`refPos`(*=Vector3r(NaN, NaN, NaN)*)
Reference position, updated at current body position each time the bound dispatcher update bounds *(auto-updated)*

`sweepLength`(*=0*)
The length used to increase the bounding boxe size, can be adjusted on the basis of previous displacement if BoundDispatcher::targetInterv>0. *(auto-updated)*

class `yade.wrapper.Aabb`(*inherits Bound → Serializable*)
Axis-aligned bounding box, for use with InsertionSortCollider. (This class is quasi-redundant since min,max are already contained in Bound itself. That might change at some point, though.)

7.2 Interactions

7.2.1 Interaction

class `yade.wrapper.Interaction`(*inherits Serializable*)
Interaction between pair of bodies.

`cellDist`
Distance of bodies in cell size units, if using periodic boundary conditions; id2 is shifted by this number of cells from its State::pos coordinates for this interaction to exist. Assigned by the collider.

> **Warning:** (internal) cellDist must survive Interaction::reset(), it is only initialized in ctor. Interaction that was cancelled by the constitutive law, was reset() and became only potential must have thepriod information if the geometric functor again makes it real. Good to know after few days of debugging that :-)

`geom`(*=uninitalized*)
Geometry part of the interaction.

id1(*=0*)
Id of the first body in this interaction.

id2(*=0*)
Id of the first body in this interaction.

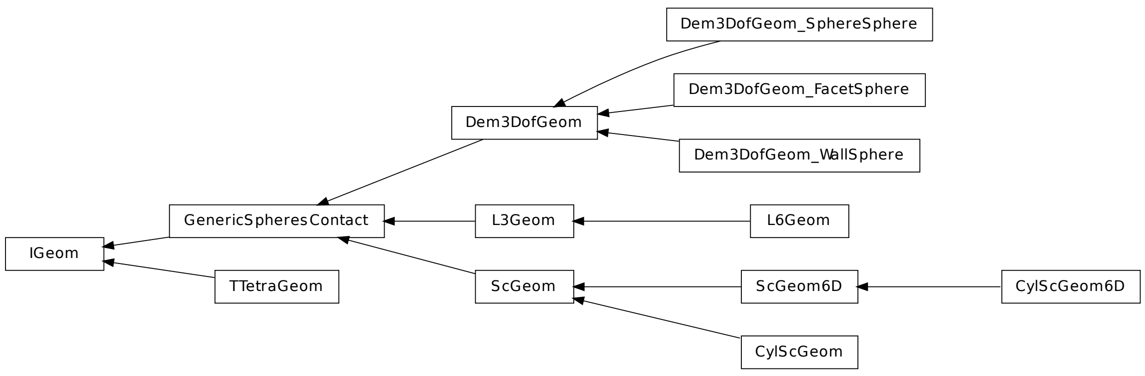
isActive
True if this interaction is active. Otherwise the forces from this interaction will not be taken into account. True by default.

isReal
True if this interaction has both geom and phys; False otherwise.

iterMadeReal(*=-1*)
Step number at which the interaction was fully (in the sense of geom and phys) created. (Should be touched only by IPhysDispatcher and InteractionLoop, therefore they are made friends of Interaction

phys(*=uninitalized*)
Physical (material) part of the interaction.

7.2.2 IGeom



class yade.wrapper.**IGeom**(*inherits Serializable*)
Geometrical configuration of interaction

dispHierarchy($\left[(bool)names=True \right]$) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

class yade.wrapper.**CylScGeom**(*inherits ScGeom → GenericSpheresContact → IGeom → Serializable*)
Geometry of a cylinder-sphere contact.

end(*=Vector3r::Zero()*)
position of 2nd node *(auto-updated)*

id3(*=0*)
id of next chained cylinder *(auto-updated)*

isDuplicate(*=0*)
this flag is turned true (1) automatically if the contact is shared between two chained cylinders. A duplicated interaction will be skipped once by the constitutive law, so that only one contact at a time is effective. If isDuplicate=2, it means one of the two duplicates has no longer geometric interaction, and should be erased by the constitutive laws.

onNode(*=false*)
: contact on node?

relPos(*=0*)
: position of the contact on the cylinder (0: node-, 1:node+) *(auto-updated)*

start(*=Vector3r::Zero()*)
: position of 1st node *(auto-updated)*

trueInt(*=-1*)
: Defines the body id of the cylinder where the contact is real, when CylScGeom::isDuplicate>0.

class yade.wrapper.CylScGeom6D(*inherits ScGeom6D → ScGeom → GenericSpheresContact → IGeom → Serializable*)

Class representing geometry of two bodies in contact. The contact has 6 DOFs (normal, 2×shear, twist, 2xbending) and uses ScGeom incremental algorithm for updating shear.

end(*=Vector3r::Zero()*)
: position of 2nd node *(auto-updated)*

id3(*=0*)
: id of next chained cylinder *(auto-updated)*

isDuplicate(*=0*)
: this flag is turned true (1) automatically if the contact is shared between two chained cylinders. A duplicated interaction will be skipped once by the constitutive law, so that only one contact at a time is effective. If isDuplicate=2, it means one of the two duplicates has no longer geometric interaction, and should be erased by the constitutive laws.

onNode(*=false*)
: contact on node?

relPos(*=0*)
: position of the contact on the cylinder (0: node-, 1:node+) *(auto-updated)*

start(*=Vector3r::Zero()*)
: position of 1st node *(auto-updated)*

trueInt(*=-1*)
: Defines the body id of the cylinder where the contact is real, when CylScGeom::isDuplicate>0.

class yade.wrapper.Dem3DofGeom(*inherits GenericSpheresContact → IGeom → Serializable*)

Abstract base class for representing contact geometry of 2 elements that has 3 degrees of freedom: normal (1 component) and shear (Vector3r, but in plane perpendicular to the normal).

displacementN() → float

displacementT() → Vector3

logCompression(*=false*)
: make strain go to -∞ for length going to zero (false by default).

refLength(*=uninitalized*)
: some length used to convert displacements to strains. *(auto-computed)*

se31(*=uninitalized*)
: Copy of body #1 se3 (needed to compute torque from the contact, strains etc). *(auto-updated)*

se32(*=uninitalized*)
: Copy of body #2 se3. *(auto-updated)*

slipToDisplacementTMax(*(float)arg2*) → float

slipToStrainTMax(*(float)arg2*) → float

strainN() → float

strainT() → Vector3

class `yade.wrapper.`**`Dem3DofGeom_FacetSphere`**(*inherits Dem3DofGeom → GenericSpheresContact → IGeom → Serializable*)

Class representing facet+sphere in contact which computes 3 degrees of freedom (normal and shear deformation).

`cp1pt`(*=uninitalized*)
: Reference contact point on the facet in facet-local coords.

`cp2rel`(*=uninitalized*)
: Orientation between +x and the reference contact point (on the sphere) in sphere-local coords

`effR2`(*=uninitalized*)
: Effective radius of sphere

`localFacetNormal`(*=uninitalized*)
: Unit normal of the facet plane in facet-local coordinates

class `yade.wrapper.`**`Dem3DofGeom_SphereSphere`**(*inherits Dem3DofGeom → GenericSpheresContact → IGeom → Serializable*)

Class representing 2 spheres in contact which computes 3 degrees of freedom (normal and shear deformation).

`cp1rel`(*=uninitalized*)
: Sphere's #1 relative orientation of the contact point with regards to sphere-local +x axis (quasi-constant)

`cp2rel`(*=uninitalized*)
: Same as cp1rel, but for sphere #2.

`effR1`(*=uninitalized*)
: Effective radius of sphere #1; can be smaller/larger than refR1 (the actual radius), but quasi-constant throughout interaction life

`effR2`(*=uninitalized*)
: Same as effR1, but for sphere #2.

class `yade.wrapper.`**`Dem3DofGeom_WallSphere`**(*inherits Dem3DofGeom → GenericSpheresContact → IGeom → Serializable*)

Representation of contact between wall and sphere, based on Dem3DofGeom.

`cp1pt`(*=uninitalized*)
: initial contact point on the wall, relative to the current contact point

`cp2rel`(*=uninitalized*)
: orientation between +x and the reference contact point (on the sphere) in sphere-local coords

`effR2`(*=uninitalized*)
: effective radius of sphere

class `yade.wrapper.`**`GenericSpheresContact`**(*inherits IGeom → Serializable*)

Class uniting ScGeom and Dem3DofGeom, for the purposes of GlobalStiffnessTimeStepper. (It might be removed inthe future). Do not use this class directly.

`contactPoint`(*=uninitalized*)
: some reference point for the interaction (usually in the middle). *(auto-computed)*

`normal`(*=uninitalized*)
: Unit vector oriented along the interaction, from particle #1, towards particle #2. *(auto-updated)*

`refR1`(*=uninitalized*)
: Reference radius of particle #1. *(auto-computed)*

`refR2`(*=uninitalized*)
: Reference radius of particle #2. *(auto-computed)*

class `yade.wrapper.`**`L3Geom`**(*inherits GenericSpheresContact → IGeom → Serializable*)

Geometry of contact given in local coordinates with 3 degress of freedom: normal and two in shear plane. [experimental]

`F`(=*Vector3r::Zero()*)
: Applied force in local coordinates [debugging only, will be removed]

`trsf`(=*Matrix3r::Identity()*)
: Transformation (rotation) from global to local coordinates. (the translation part is in GenericSpheresContact.contactPoint)

`u`(=*Vector3r::Zero()*)
: Displacement components, in local coordinates. *(auto-updated)*

`u0`
: Zero displacement value; u0 should be always subtracted from the *geometrical* displacement *u* computed by appropriate IGeomFunctor, resulting in *u*. This value can be changed for instance

1. by IGeomFunctor, e.g. to take in account large shear displacement value unrepresentable by underlying geomeric algorithm based on quaternions)
2. by LawFunctor, to account for normal equilibrium position different from zero geometric overlap (set once, just after the interaction is created)
3. by LawFunctor to account for plastic slip.

Note: Never set an absolute value of *u0*, only increment, since both IGeomFunctor and LawFunctor use it. If you need to keep track of plastic deformation, store it in IPhys isntead (this might be changed: have *u0* for LawFunctor exclusively, and a separate value stored (when that is needed) inside classes deriving from L3Geom.

class `yade.wrapper.L6Geom`(*inherits L3Geom → GenericSpheresContact → IGeom → Serializable*)
: Geometric of contact in local coordinates with 6 degrees of freedom. [experimental]

`phi`(=*Vector3r::Zero()*)
: Rotation components, in local coordinates. *(auto-updated)*

`phi0`(=*Vector3r::Zero()*)
: Zero rotation, should be always subtracted from *phi* to get the value. See L3Geom.u0.

class `yade.wrapper.ScGeom`(*inherits GenericSpheresContact → IGeom → Serializable*)
: Class representing geometry of a contact point between two bodies with a non-spherical bodies (Facet, Plane, Box, ChainedCylinder), or between non-spherical bodies. The contact has 3 DOFs (normal and 2×shear) and uses incremental algorithm for updating shear.

We use symbols $\mathbf{x}$, $\mathbf{v}$, $\boldsymbol{\omega}$ respectively for position, linear and angular velocities (all in global coordinates) and r for particles radii; subscripted with 1 or 2 to distinguish 2 spheres in contact. Then we compute unit contact normal

$$\mathbf{n} = \frac{\mathbf{x}_2 - \mathbf{x}_1}{||\mathbf{x}_2 - \mathbf{x}_1||}$$

Relative velocity of spheres is then

$$\mathbf{v}_{12} = (\mathbf{v}_2 + \boldsymbol{\omega}_2 \times (-r_2\mathbf{n})) - (\mathbf{v}_1 + \boldsymbol{\omega}_1 \times (r_1\mathbf{n}))$$

and its shear component

$$\Delta\mathbf{v}_{12}^s = \mathbf{v}_{12} - (\mathbf{n} \cdot \mathbf{v}_{12})\mathbf{n}.$$

Tangential displacement increment over last step then reads

$$\mathbf{x}_{12}^s = \Delta t \mathbf{v}_{12}^s.$$

`incidentVel`(*(Interaction)i*[, *(bool)avoidGranularRatcheting=True*]) → Vector3
: Return incident velocity of the interaction.

penetrationDepth(*=NaN*)
Penetration distance of spheres (positive if overlapping)

relAngVel(*(Interaction)i*) → Vector3
Return relative angular velocity of the interaction.

shearInc(*=Vector3r::Zero()*)
Shear displacement increment in the last step

class yade.wrapper.**ScGeom6D**(*inherits ScGeom → GenericSpheresContact → IGeom → Serializable*)
Class representing geometry of two bodies in contact. The contact has 6 DOFs (normal, 2×shear, twist, 2xbending) and uses ScGeom incremental algorithm for updating shear.

bending(*=Vector3r::Zero()*)
Bending at contact as a vector defining axis of rotation and angle (angle=norm).

initialOrientation1(*=Quaternionr(1.0, 0.0, 0.0, 0.0)*)
Orientation of body 1 one at initialisation time *(auto-updated)*

initialOrientation2(*=Quaternionr(1.0, 0.0, 0.0, 0.0)*)
Orientation of body 2 one at initialisation time *(auto-updated)*

twist(*=0*)
Elastic twist angle of the contact.

twistCreep(*=Quaternionr(1.0, 0.0, 0.0, 0.0)*)
Stored creep, substracted from total relative rotation for computation of elastic moment *(auto-updated)*

class yade.wrapper.**TTetraGeom**(*inherits IGeom → Serializable*)
Geometry of interaction between 2 tetrahedra, including volumetric characteristics

contactPoint(*=uninitalized*)
Contact point (global coords)

equivalentCrossSection(*=NaN*)
Cross-section of the overlap (perpendicular to the axis of least inertia

equivalentPenetrationDepth(*=NaN*)
??

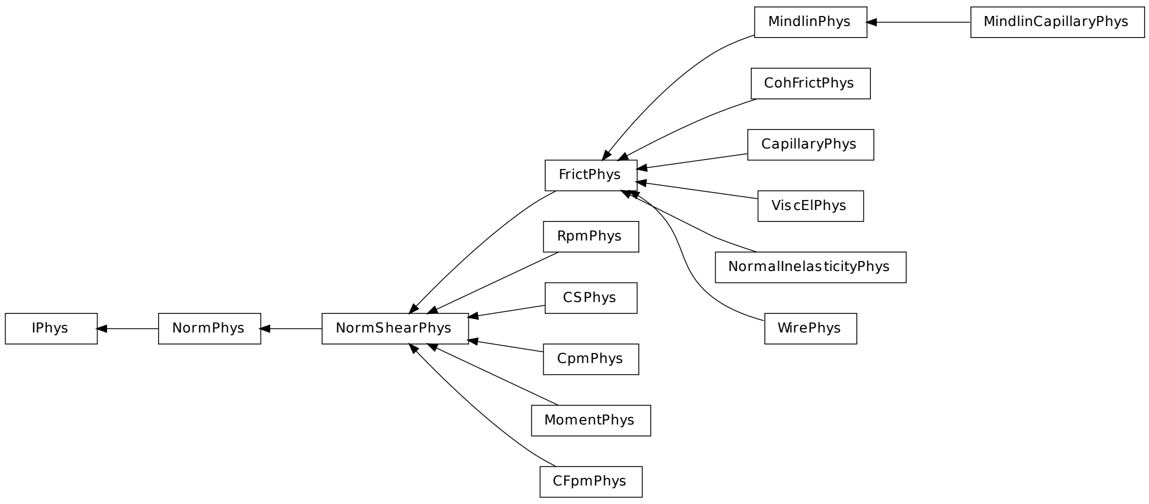
maxPenetrationDepthA(*=NaN*)
??

maxPenetrationDepthB(*=NaN*)
??

normal(*=uninitalized*)
Normal of the interaction, directed in the sense of least inertia of the overlap volume

penetrationVolume(*=NaN*)
Volume of overlap [m^3]

7.2.3 IPhys



class yade.wrapper.**IPhys**(*inherits Serializable*)
Physical (material) properties of interaction.

dispHierarchy([*(bool)names=True*]) → list
Return list of dispatch classes (from down upwards), starting with the class instance itself, top-level indexable at last. If names is true (default), return class names rather than numerical indices.

dispIndex
Return class index of this instance.

class yade.wrapper.**CFpmPhys**(*inherits NormShearPhys → NormPhys → IPhys → Serializable*)
Representation of a single interaction of the CFpm type, storage for relevant parameters

FnMax(*=0*)
Defines the maximum admissible normal force in traction FnMax=tensileStrength*crossSection, with crossSection=pi*Rmin^2. [Pa]

FsMax(*=0*)
Defines the maximum admissible tangential force in shear FsMax=cohesion*FnMax, with crossSection=pi*Rmin^2. [Pa]

cumulativeRotation(*=0*)
Cumulated rotation... [-]

frictionAngle(*=0*)
defines Coulomb friction. [deg]

initD(*=0*)
equilibrium distance for particles. Computed as the initial interparticular distance when bonded particle interact. initD=0 for non cohesive interactions.

initialOrientation1(*=Quaternionr(1.0, 0.0, 0.0, 0.0)*)
Used for moment computation.

initialOrientation2(*=Quaternionr(1.0, 0.0, 0.0, 0.0)*)
Used for moment computation.

isCohesive(*=false*)
If false, particles interact in a frictional way. If true, particles are bonded regarding the given cohesion and tensileStrength.

kr(*=0*)
Defines the stiffness to compute the resistive moment in rotation. [-]

maxBend(*=0*)
Defines the maximum admissible resistive moment in rotation Mtmax=maxBend*Fn, maxBend=eta*meanRadius. [m]

moment_bending(*=Vector3r::Zero()*)
[N.m]

moment_twist(*=Vector3r::Zero()*)
[N.m]

prevNormal(*=Vector3r::Zero()*)
Normal to the contact at previous time step.

strengthSoftening(*=0*)
Defines the softening when Dtensile is reached to avoid explosion. Typically, when D > Dtensile, Fn=FnMax - (kn/strengthSoftening)*(Dtensile-D). [-]

tanFrictionAngle(*=0*)
Tangent of frictionAngle. [-]

class yade.wrapper.**CSPhys**(*inherits NormShearPhys → NormPhys → IPhys → Serializable*)
Physical properties for Cundall&Strack constitutive law, created by Ip2_2xFrictMat_CSPhys.

frictionAngle(*=NaN*)
Friction angle of the interaction. *(auto-computed)*

tanFrictionAngle(*=NaN*)
Precomputed tangent of CSPhys::frictionAngle. *(auto-computed)*

class yade.wrapper.**CapillaryPhys**(*inherits FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)
Physics (of interaction) for Law2_ScGeom_CapillaryPhys_Capillarity.

CapillaryPressure(*=0.*)
Value of the capillary pressure Uc defines as Ugas-Uliquid

Delta1(*=0.*)
Defines the surface area wetted by the meniscus on the smallest grains of radius R1 (R1<R2)

Delta2(*=0.*)
Defines the surface area wetted by the meniscus on the biggest grains of radius R2 (R1<R2)

Fcap(*=Vector3r::Zero()*)
Capillary Force produces by the presence of the meniscus

Vmeniscus(*=0.*)
Volume of the menicus

fusionNumber(*=0.*)
Indicates the number of meniscii that overlap with this one

isBroken(*=false*)
If true, capillary force is zero and liquid bridge is inactive.

meniscus(*=false*)
Presence of a meniscus if true

class yade.wrapper.**CohFrictPhys**(*inherits FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)

cohesionBroken(*=true*)
is cohesion active? will be set false when a fragile contact is broken

cohesionDisablesFriction(*=false*)
is shear strength the sum of friction and adhesion or only adhesion?

creep_viscosity(*=-1*)
creep viscosity [Pa.s/m].

fragile(*=true*)
do cohesion disapear when contact strength is exceeded?

`kr`(*=0*)
: rotational stiffness [N.m/rad]

`ktw`(*=0*)
: twist stiffness [N.m/rad]

`maxRollPl`(*=0.0*)
: Coefficient to determine the maximum plastic rolling moment.

`maxTwistMoment`(*=Vector3r::Zero()*)
: Maximum elastic value for the twisting moment (if zero, plasticity will not be applied). In CohFrictMat a parameter should be added to decide what value should be attributed to this threshold value.

`momentRotationLaw`(*=false*)
: use bending/twisting moment at contacts. See CohFrictPhys::cohesionDisablesFriction for details.

`moment_bending`(*=Vector3r(0, 0, 0)*)
: Bending moment

`moment_twist`(*=Vector3r(0, 0, 0)*)
: Twist moment

`normalAdhesion`(*=0*)
: tensile strength

`shearAdhesion`(*=0*)
: cohesive part of the shear strength (a frictional term might be added depending on Law2_-ScGeom6D_CohFrictPhys_CohesionMoment::always_use_moment_law)

`unp`(*=0*)
: plastic normal displacement, only used for tensile behaviour and if CohFrictPhys::fragile‘=false. :ydefault:‘0

`unpMax`(*=0*)
: maximum value of plastic normal displacement, after that the interaction breaks even if CohFrictPhys::fragile‘=false. The default value (0) means no maximum. :ydefault:‘0

class `yade.wrapper.`**`CpmPhys`**(*inherits NormShearPhys → NormPhys → IPhys → Serializable*)
: Representation of a single interaction of the Cpm type: storage for relevant parameters.

  Evolution of the contact is governed by Law2_Dem3DofGeom_CpmPhys_Cpm, that includes damage effects and chages of parameters inside CpmPhys. See *cpm-model* for details.

  `E`(*=NaN*)
  : normal modulus (stiffness / crossSection) [Pa]

  `Fn`
  : Magnitude of normal force.

  `Fs`
  : Magnitude of shear force

  `G`(*=NaN*)
  : shear modulus [Pa]

  `crackOpening`(*=NaN*)
  : Crack opening (extansion of the bond) when the bond is fully broken in tension. [m]

  `crossSection`(*=NaN*)
  : equivalent cross-section associated with this contact [m^2]

  `cummBetaCount` = 0

  `cummBetaIter` = 0

  `damLaw`(*=-1*)
  : Law for softening part of uniaxial tension. 0 for linear, 1 for exponential

dmgOverstress(*=0*)
: damage viscous overstress (at previous step or at current step)

dmgRateExp(*=0*)
: exponent in the rate-dependent damage evolution

dmgStrain(*=0*)
: damage strain (at previous or current step)

dmgTau(*=-1*)
: characteristic time for damage (if non-positive, the law without rate-dependence is used)

epsCrackOnset(*=NaN*)
: strain at which the material starts to behave non-linearly

epsFracture(*=NaN*)
: strain at which the bond is fully broken [-]

epsN
: Current normal strain

epsNPl(*=0*)
: normal plastic strain (initially zero)

epsPlSum(*=0*)
: cummulative shear plastic strain measure (scalar) on this contact

epsT(*=Vector3r::Zero()*)
: Total shear strain (either computed from increments with ScGeom or simple copied with Dem3DofGeom) *(auto-updated)*

epsTrans(*=0*)
: Transversal strain (perpendicular to the contact axis)

isCohesive(*=false*)
: if not cohesive, interaction is deleted when distance is greater than zero.

isoPrestress(*=0*)
: “prestress” of this link (used to simulate isotropic stress)

kappaD(*=0*)
: Up to now maximum normal strain (semi-norm), non-decreasing in time.

neverDamage(*=false*)
: the damage evolution function will always return virgin state

omega
: Damage internal variable

plRateExp(*=0*)
: exponent in the rate-dependent viscoplasticity

plTau(*=-1*)
: characteristic time for viscoplasticity (if non-positive, no rate-dependence for shear)

relDuctility(*=NaN*)
: Relative ductility of bonds in normal direction

relResidualStrength
: Relative residual strength

sigmaN
: Current normal stress

sigmaT
: Current shear stress

tanFrictionAngle(*=NaN*)
: tangens of internal friction angle [-]

undamagedCohesion(*=NaN*)
: virgin material cohesion [Pa]

class yade.wrapper.**FrictPhys**(*inherits NormShearPhys → NormPhys → IPhys → Serializable*)
The simple linear elastic-plastic interaction with friction angle, like in the traditional [CundallStrack1979]

tangensOfFrictionAngle(*=NaN*)
: tan of angle of friction

class yade.wrapper.**MindlinCapillaryPhys**(*inherits MindlinPhys → FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)
Adds capillary physics to Mindlin's interaction physics.

CapillaryPressure(*=0.*)
: Value of the capillary pressure Uc defines as Ugas-Uliquid

Delta1(*=0.*)
: Defines the surface area wetted by the meniscus on the smallest grains of radius R1 (R1<R2)

Delta2(*=0.*)
: Defines the surface area wetted by the meniscus on the biggest grains of radius R2 (R1<R2)

Fcap(*=Vector3r::Zero()*)
: Capillary Force produces by the presence of the meniscus

Vmeniscus(*=0.*)
: Volume of the menicus

fusionNumber(*=0.*)
: Indicates the number of meniscii that overlap with this one

isBroken(*=false*)
: If true, capillary force is zero and liquid bridge is inactive.

meniscus(*=false*)
: Presence of a meniscus if true

class yade.wrapper.**MindlinPhys**(*inherits FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)
Representation of an interaction of the Hertz-Mindlin type.

Fs(*=Vector2r::Zero()*)
: Shear force in local axes (computed incrementally)

adhesionForce(*=0.0*)
: Force of adhesion as predicted by DMT

alpha(*=0.0*)
: Constant coefficient to define contact viscous damping for non-linear elastic force-displacement relationship.

betan(*=0.0*)
: Fraction of the viscous damping coefficient (normal direction) equal to $\frac{c_n}{C_{n,crit}}$.

betas(*=0.0*)
: Fraction of the viscous damping coefficient (shear direction) equal to $\frac{c_s}{C_{s,crit}}$.

isAdhesive(*=false*)
: bool to identify if the contact is adhesive, that is to say if the contact force is attractive

isSliding(*=false*)
: check if the contact is sliding (useful to calculate the ratio of sliding contacts)

kno(*=0.0*)
: Constant value in the formulation of the normal stiffness

kr(*=0.0*)
: Rotational stiffness

kso(=*0.0*)
Constant value in the formulation of the tangential stiffness

ktw(=*0.0*)
Rotational stiffness

maxBendPl(=*0.0*)
Coefficient to determine the maximum plastic moment to apply at the contact

momentBend(=*Vector3r::Zero()*)
Artificial bending moment to provide rolling resistance in order to account for some degree of interlocking between particles

momentTwist(=*Vector3r::Zero()*)
Artificial twisting moment (no plastic condition can be applied at the moment)

normalViscous(=*Vector3r::Zero()*)
Normal viscous component

prevU(=*Vector3r::Zero()*)
Previous local displacement; only used with Law2_L3Geom_FrictPhys_HertzMindlin.

radius(=*NaN*)
Contact radius (only computed with Law2_ScGeom_MindlinPhys_Mindlin::calcEnergy)

shearElastic(=*Vector3r::Zero()*)
Total elastic shear force

shearViscous(=*Vector3r::Zero()*)
Shear viscous component

usElastic(=*Vector3r::Zero()*)
Total elastic shear displacement (only elastic part)

usTotal(=*Vector3r::Zero()*)
Total elastic shear displacement (elastic+plastic part)

class yade.wrapper.**MomentPhys**(*inherits NormShearPhys → NormPhys → IPhys → Serializable*)
Physical interaction properties for use with Law2_SCG_MomentPhys_CohesionlessMomentRotation, created by Ip2_MomentMat_MomentMat_MomentPhys.

Eta(=*0*)
??

cumulativeRotation(=*0*)
??

frictionAngle(=*0*)
Friction angle [rad]

initialOrientation1(=*Quaternionr::Identity()*)
??

initialOrientation2(=*Quaternionr::Identity()*)
??

kr(=*0*)
rolling stiffness

moment_bending(=*Vector3r::Zero()*)
??

moment_twist(=*Vector3r::Zero()*)
??

prevNormal(=*Vector3r::Zero()*)
Normal in the previous step.

shear(=*Vector3r::Zero()*)
??

tanFrictionAngle(*=0*)
Tangent of friction angle

class yade.wrapper.**NormPhys**(*inherits IPhys → Serializable*)
Abstract class for interactions that have normal stiffness.

kn(*=0*)
Normal stiffness

normalForce(*=Vector3r::Zero()*)
Normal force after previous step (in global coordinates).

class yade.wrapper.**NormShearPhys**(*inherits NormPhys → IPhys → Serializable*)
Abstract class for interactions that have shear stiffnesses, in addition to normal stiffness. This class is used in the PFC3d-style stiffness timestepper.

ks(*=0*)
Shear stiffness

shearForce(*=Vector3r::Zero()*)
Shear force after previous step (in global coordinates).

class yade.wrapper.**NormalInelasticityPhys**(*inherits FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)
Physics (of interaction) for using Law2_ScGeom6D_NormalInelasticityPhys_NormalInelasticity : with inelastic unloadings

forMaxMoment(*=1.0*)
parameter stored for each interaction, and allowing to compute the maximum value of the exchanged torque : TorqueMax= forMaxMoment * NormalForce

knLower(*=0.0*)
the stifness corresponding to a virgin load for example

kr(*=0.0*)
the rolling stiffness of the interaction

moment_bending(*=Vector3r(0, 0, 0)*)
Bending moment. Defined here, being initialized as it should be, to be used in Law2_ScGeom6D_NormalInelasticityPhys_NormalInelasticity

moment_twist(*=Vector3r(0, 0, 0)*)
Twist moment. Defined here, being initialized as it should be, to be used in Law2_ScGeom6D_NormalInelasticityPhys_NormalInelasticity

previousFn(*=0.0*)
the value of the normal force at the last time step

previousun(*=0.0*)
the value of this un at the last time step

unMax(*=0.0*)
the maximum value of penetration depth of the history of this interaction

class yade.wrapper.**RpmPhys**(*inherits NormShearPhys → NormPhys → IPhys → Serializable*)
Representation of a single interaction of the Cpm type: storage for relevant parameters.

Evolution of the contact is governed by Law2_Dem3DofGeom_CpmPhys_Cpm, that includes damage effects and chages of parameters inside CpmPhys

E(*=NaN*)
normal modulus (stiffness / crossSection) [Pa]

G(*=NaN*)
shear modulus [Pa]

crossSection(*=0*)
equivalent cross-section associated with this contact [m^2]

isCohesive(*=false*)
: if not cohesive, interaction is deleted when distance is greater than lengthMaxTension or less than lengthMaxCompression.

lengthMaxCompression(*=0*)
: Maximal penetration of particles during compression. If it is more, the interaction is deleted [m]

lengthMaxTension(*=0*)
: Maximal distance between particles during tension. If it is more, the interaction is deleted [m]

tanFrictionAngle(*=NaN*)
: tangens of internal friction angle [-]

class yade.wrapper.**ViscElPhys**(*inherits FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)
IPhys created from ViscElMat, for use with Law2_ScGeom_ViscElPhys_Basic.

cn(*=NaN*)
: Normal viscous constant

cs(*=NaN*)
: Shear viscous constant

class yade.wrapper.**WirePhys**(*inherits FrictPhys → NormShearPhys → NormPhys → IPhys → Serializable*)
Representation of a single interaction of the WirePM type, storage for relevant parameters

displForceValues(*=uninitalized*)
: Defines the values for force-displacement curve.

initD(*=0*)
: Equilibrium distance for particles. Computed as the initial inter-particular distance when particle are linked.

isDoubleTwist(*=false*)
: If true the properties of the interaction will be defined as a double-twisted wire.

isLinked(*=false*)
: If true particles are linked and will interact. Interactions are linked automatically by the definition of the corresponding interaction radius. The value is false if the wire breaks (no more interaction).

limitFactor(*=0.*)
: This value indicates on how far from failing the wire is, e.g. actual normal displacement divided by admissible normal displacement multiplied by actual normal force divided by admissible normal force.

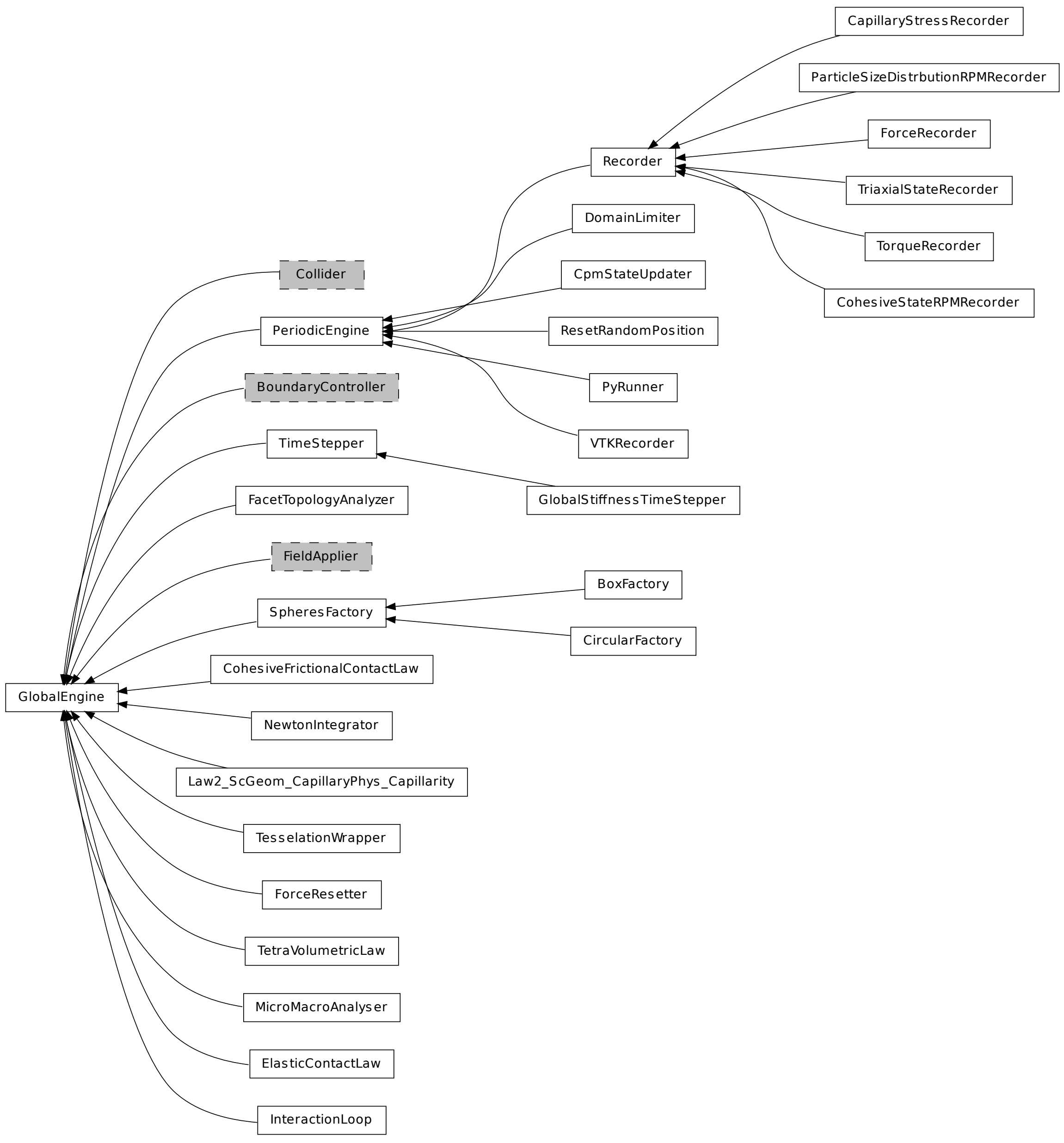
plastD
: Plastic part of the inter-particular distance of the previous step.

Note: Only elastic displacements are reversible (the elastic stiffness is used for unloading) and compressive forces are inadmissible. The compressive stiffness is assumed to be equal to zero (see [Bertrand2005]).

stiffnessValues(*=uninitalized*)
: Defines the values for the different stiffness (first value corresponds to elastic stiffness kn).

7.3 Global engines

7.3.1 GlobalEngine



class `yade.wrapper.GlobalEngine`(*inherits Engine → Serializable*)
: Engine that will generally affect the whole simulation (contrary to PartialEngine).

class `yade.wrapper.BoxFactory`(*inherits SpheresFactory → GlobalEngine → Engine → Serializable*)
: Box geometry of the SpheresFactory region, given by extents and center

  `center`(*=Vector3r(NaN, NaN, NaN)*)
  : Center of the region

  `extents`(*=Vector3r(NaN, NaN, NaN)*)
  : Extents of the region

class `yade.wrapper.CapillaryStressRecorder`(*inherits Recorder → PeriodicEngine → GlobalEngine → Engine → Serializable*)

Records information from capillary meniscii on samples submitted to triaxial compressions. -> New formalism needs to be tested!!!

class `yade.wrapper.CircularFactory`(*inherits SpheresFactory → GlobalEngine → Engine → Serializable*)

Circular geometry of the SpheresFactory region. It can be disk (given by radius and center), or cylinder (given by radius, length and center).

`center`(*=Vector3r(NaN, NaN, NaN)*)
: Center of the region

`length`(*=0*)
: Length of the cylindrical region (0 by default)

`radius`(*=NaN*)
: Radius of the region

class `yade.wrapper.CohesiveFrictionalContactLaw`(*inherits GlobalEngine → Engine → Serializable*)

[DEPRECATED] Loop over interactions applying Law2_ScGeom6D_CohFrictPhys_CohesionMoment on all interactions.

Note: Use InteractionLoop and Law2_ScGeom6D_CohFrictPhys_CohesionMoment instead of this class for performance reasons.

`always_use_moment_law`(*=false*)
: If true, use bending/twisting moments at all contacts. If false, compute moments only for cohesive contacts.

`creep_viscosity`(*=false*)
: creep viscosity [Pa.s/m]. probably should be moved to Ip2_CohFrictMat_CohFrictMat_-CohFrictPhys...

`neverErase`(*=false*)
: Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. Law2_ScGeom_CapillaryPhys_Capillarity)

`shear_creep`(*=false*)
: activate creep on the shear force, using CohesiveFrictionalContactLaw::creep_viscosity.

`twist_creep`(*=false*)
: activate creep on the twisting moment, using CohesiveFrictionalContactLaw::creep_viscosity.

class `yade.wrapper.CohesiveStateRPMRecorder`(*inherits Recorder → PeriodicEngine → GlobalEngine → Engine → Serializable*)

Store number of cohesive contacts in RPM model to file.

`numberCohesiveContacts`(*=0*)
: Number of cohesive contacts found at last run. [-]

class `yade.wrapper.CpmStateUpdater`(*inherits PeriodicEngine → GlobalEngine → Engine → Serializable*)

Update CpmState of bodies based on state variables in CpmPhys of interactions with this bod. In particular, bodies' colors and CpmState::normDmg depending on average damage of their interactions and number of interactions that were already fully broken and have disappeared is updated. This engine contains its own loop (2 loops, more precisely) over all bodies and should be run periodically to update colors during the simulation, if desired.

`avgRelResidual`(*=NaN*)
: Average residual strength at last run.

`maxOmega`(*=NaN*)
: Globally maximum damage parameter at last run.

class `yade.wrapper.DomainLimiter`(*inherits PeriodicEngine → GlobalEngine → Engine → Serializable*)

Delete particles that are out of axis-aligned box given by *lo* and *hi*.

hi(=*Vector3r(0, 0, 0)*)
: Upper corner of the domain.

lo(=*Vector3r(0, 0, 0)*)
: Lower corner of the domain.

mDeleted(=*0*)
: Mass of deleted particles.

mask(=*-1*)
: If mask is defined, only particles with corresponding groupMask will be deleted.

nDeleted(=*0*)
: Cummulative number of particles deleted.

vDeleted(=*0*)
: Volume of deleted particles.

class yade.wrapper.**ElasticContactLaw**(*inherits GlobalEngine → Engine → Serializable*)
[DEPRECATED] Loop over interactions applying Law2_ScGeom_FrictPhys_CundallStrack on all interactions.

Note: Use InteractionLoop and Law2_ScGeom_FrictPhys_CundallStrack instead of this class for performance reasons.

neverErase(=*false*)
: Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. Law2_ScGeom_CapillaryPhys_Capillarity)

class yade.wrapper.**FacetTopologyAnalyzer**(*inherits GlobalEngine → Engine → Serializable*)
Initializer for filling adjacency geometry data for facets.

Common vertices and common edges are identified and mutual angle between facet faces is written to Facet instances. If facets don't move with respect to each other, this must be done only at the beginng.

commonEdgesFound(=*0*)
: how many common edges were identified during last run. *(auto-updated)*

commonVerticesFound(=*0*)
: how many common vertices were identified during last run. *(auto-updated)*

projectionAxis(=*Vector3r::UnitX()*)
: Axis along which to do the initial vertex sort

relTolerance(=*1e-4*)
: maximum distance of 'identical' vertices, relative to minimum facet size

class yade.wrapper.**ForceRecorder**(*inherits Recorder → PeriodicEngine → GlobalEngine → Engine → Serializable*)
Engine saves the resultant force affecting to bodies, listed in *ids*. For instance, can be useful for defining the forces, which affects to _buldozer_ during its work.

ids(=*uninitalized*)
: List of bodies whose state will be measured

totalForce(=*Vector3r::Zero()*)
: Resultant force, returning by the function.

class yade.wrapper.**ForceResetter**(*inherits GlobalEngine → Engine → Serializable*)
Reset all forces stored in Scene::forces (`O.forces` in python). Typically, this is the first engine to be run at every step. In addition, reset those energies that should be reset, if energy tracing is enabled.

class yade.wrapper.**GlobalStiffnessTimeStepper**(*inherits TimeStepper → GlobalEngine → Engine → Serializable*)
An engine assigning the time-step as a fraction of the minimum eigen-period in the problem

`defaultDt`(*=1*)
: used as the initial value of the timestep (especially useful in the first steps when no contact exist). Can be defined using utils.PWaveTimeStep

`maxDt`(*=1*)
: used as max value of the timestep whatever the computed value

`previousDt`(*=1*)
: last computed dt *(auto-updated)*

`timestepSafetyCoefficient`(*=0.8*)
: safety factor between the minimum eigen-period and the final assigned dt (less than 1))

class `yade.wrapper.`**`InteractionLoop`**(*inherits GlobalEngine → Engine → Serializable*)
Unified dispatcher for handling interaction loop at every step, for parallel performance reasons.

Special constructor

Constructs from 3 lists of Ig2, Ip2, Law functors respectively; they will be passed to interal dispatchers, which you might retrieve.

`callbacks`(*=uninitalized*)
: Callbacks which will be called for every Interaction, if activated.

`eraseIntsInLoop`(*=false*)
: Defines if the interaction loop should erase pending interactions, else the collider takes care of that alone (depends on what collider is used).

`geomDispatcher`(*=new IGeomDispatcher*)
: IGeomDispatcher object that is used for dispatch.

`lawDispatcher`(*=new LawDispatcher*)
: LawDispatcher object used for dispatch.

`physDispatcher`(*=new IPhysDispatcher*)
: IPhysDispatcher object used for dispatch.

class `yade.wrapper.`**`Law2_ScGeom_CapillaryPhys_Capillarity`**(*inherits GlobalEngine → Engine → Serializable*)
This law allows one to take into account capillary forces/effects between spheres coming from the presence of interparticular liquid bridges (menisci).

The control parameter is the capillary pressure (or suction) Uc = ugas - Uliquid. Liquid bridges properties (volume V, extent over interacting grains delta1 and delta2) are computed as a result of the defined capillary pressure and of the interacting geometry (spheres radii and interparticular distance).

References: in english [Scholtes2009b]; more detailed, but in french [Scholtes2009d].

The law needs ascii files M(r=i) with i=R1/R2 to work (see https://yade-dem.org/index.php/CapillaryTriaxialTest). These ASCII files contain a set of results from the resolution of the Laplace-Young equation for different configurations of the interacting geometry.

In order to allow capillary forces between distant spheres, it is necessary to enlarge the bounding boxes using Bo1_Sphere_Aabb::aabbEnlargeFactor and make the Ig2 define define distant interactions via interactionDetectionFactor. It is also necessary to disable interactions removal by the constitutive law (Law2). The only combinations of laws supported are currently capillary law + Law2_ScGeom_FrictPhys_CundallStrack and capillary law + Law2_ScGeom_MindlinPhys_Mindlin (and the other variants of Hertz-Mindlin).

See CapillaryPhys-example.py for an example script.

`CapillaryPressure`(*=0.*)
: Value of the capillary pressure Uc defines as Uc=Ugas-Uliquid

`binaryFusion`(*=true*)
: If true, capillary forces are set to zero as soon as, at least, 1 overlap (menisci fusion) is detected

`createDistantMeniscii`(*=false*)
: Generate meniscii between distant spheres? Else only maintain the existing one. For modeling a wetting path this flag should always be false. For a drying path it should be true for one step (initialization) then false, as in the logic of [Scholtes2009c]

`fusionDetection`(*=false*)
: If true potential menisci overlaps are checked

`hertzOn`(*=false*)
: *(auto-updated)* true if hertz model is used

class `yade.wrapper.`**`MicroMacroAnalyser`**(*inherits GlobalEngine → Engine → Serializable*)
: Compute fabric tensor, local porosity, local deformation, and other micromechanicaly defined quantities based on triangulation/tesselation of the packing.

`compDeformation`(*=false*)
: Is the engine just saving states or also computing and outputing deformations for each increment?

`compIncrt`(*=false*)
: Should increments of force and displacements be defined on [n,n+1]? If not, states will be saved with only positions and forces (no displacements).

`incrtNumber`(*=1*)

`interval`(*=100*)
: Number of timesteps between analyzed states.

`nonSphereAsFictious`(*=true*)
: bodies that are not spheres will be used to defines bounds (else just skipped).

`outputFile`(*="MicroMacroAnalysis"*)
: Base name for increment analysis output file.

`stateFileName`(*="state"*)
: Base name of state files.

`stateNumber`(*=0*)
: A number incremented and appended at the end of output files to reflect increment number.

class `yade.wrapper.`**`NewtonIntegrator`**(*inherits GlobalEngine → Engine → Serializable*)
: Engine integrating newtonian motion equations.

`damping`(*=0.2*)
: damping coefficient for Cundall's non viscous damping (see [Chareyre2005]) [-]

`exactAsphericalRot`(*=true*)
: Enable more exact body rotation integrator for aspherical bodies *only*, using formulation from [Allen1989], pg. 89.

`gravity`(*=Vector3r::Zero()*)
: Gravitational acceleration (effectifely replaces GravityEngine).

`kinSplit`(*=false*)
: Whether to separately track translational and rotational kinetic energy.

`maxVelocitySq`(*=NaN*)
: store square of max. velocity, for informative purposes; computed again at every step. *(auto-updated)*

`prevVelGrad`(*=Matrix3r::Zero()*)
: Store previous velocity gradient (Cell::velGrad) to track acceleration. *(auto-updated)*

`warnNoForceReset`(*=true*)
: Warn when forces were not resetted in this step by ForceResetter; this mostly points to ForceResetter being forgotten incidentally and should be disabled only with a good reason.

class `yade.wrapper.ParticleSizeDistrbutionRPMRecorder`(*inherits Recorder → PeriodicEngine → GlobalEngine → Engine → Serializable*)

Store number of PSD in RPM model to file.

numberCohesiveContacts(*=0*)
Number of cohesive contacts found at last run. [-]

class `yade.wrapper.PeriodicEngine`(*inherits GlobalEngine → Engine → Serializable*)

Run Engine::action with given fixed periodicity real time (=wall clock time, computation time), virtual time (simulation time), iteration number), by setting any of those criteria (virtPeriod, realPeriod, iterPeriod) to a positive value. They are all negative (inactive) by default.

The number of times this engine is activated can be limited by setting nDo>0. If the number of activations will have been already reached, no action will be called even if an active period has elapsed.

If initRun is set (false by default), the engine will run when called for the first time; otherwise it will only start counting period (realLast etc interal variables) from that point, but without actually running, and will run only once a period has elapsed since the initial run.

This class should not be used directly; rather, derive your own engine which you want to be run periodically.

Derived engines should override Engine::action(), which will be called periodically. If the derived Engine overrides also Engine::isActivated, it should also take in account return value from PeriodicEngine::isActivated, since otherwise the periodicity will not be functional.

Example with PyRunner, which derives from PeriodicEngine; likely to be encountered in python scripts:

```
PyRunner(realPeriod=5,iterPeriod=10000,command='print O.iter')
```

will print iteration number every 10000 iterations or every 5 seconds of wall clock time, whiever comes first since it was last run.

initRun(*=false*)
Run the first time we are called as well.

iterLast(*=0*)
Tracks step number of last run *(auto-updated)*.

iterPeriod(*=0, deactivated*)
Periodicity criterion using step number (deactivated if <= 0)

nDo(*=-1, deactivated*)
Limit number of executions by this number (deactivated if negative)

nDone(*=0*)
Track number of executions (cummulative) *(auto-updated)*.

realLast(*=0*)
Tracks real time of last run *(auto-updated)*.

realPeriod(*=0, deactivated*)
Periodicity criterion using real (wall clock, computation, human) time (deactivated if <=0)

virtLast(*=0*)
Tracks virtual time of last run *(auto-updated)*.

virtPeriod(*=0, deactivated*)
Periodicity criterion using virtual (simulation) time (deactivated if <= 0)

class `yade.wrapper.PyRunner`(*inherits PeriodicEngine → GlobalEngine → Engine → Serializable*)

Execute a python command periodically, with defined (and adjustable) periodicity. See PeriodicEngine documentation for details.

command(*=”“*)
Command to be run by python interpreter. Not run if empty.

class `yade.wrapper.Recorder`(*inherits PeriodicEngine → GlobalEngine → Engine → Serializable*)

Engine periodically storing some data to (one) external file. In addition PeriodicEngine, it handles opening the file as needed. See PeriodicEngine for controlling periodicity.

`addIterNum`(*=false*)
: Adds an iteration number to the file name, when the file was created. Useful for creating new files at each call (false by default)

`file`(*=uninitalized*)
: Name of file to save to; must not be empty.

`truncate`(*=false*)
: Whether to delete current file contents, if any, when opening (false by default)

class `yade.wrapper.ResetRandomPosition`(*inherits PeriodicEngine → GlobalEngine → Engine → Serializable*)

Creates spheres during simulation, placing them at random positions. Every time called, one new sphere will be created and inserted in the simulation.

`angularVelocity`(*=Vector3r::Zero()*)
: Mean angularVelocity of spheres.

`angularVelocityRange`(*=Vector3r::Zero()*)
: Half size of a angularVelocity distribution interval. New sphere will have random angularVelocity within the range angularVelocity±angularVelocityRange.

`factoryFacets`(*=uninitalized*)
: The geometry of the section where spheres will be placed; they will be placed on facets or in volume between them depending on *volumeSection* flag.

`maxAttempts`(*=20*)
: Max attempts to place sphere. If placing the sphere in certain random position would cause an overlap with any other physical body in the model, SpheresFactory will try to find another position.

`normal`(*=Vector3r(0, 1, 0)*)
: ??

`point`(*=Vector3r::Zero()*)
: ??

`subscribedBodies`(*=uninitalized*)
: Affected bodies.

`velocity`(*=Vector3r::Zero()*)
: Mean velocity of spheres.

`velocityRange`(*=Vector3r::Zero()*)
: Half size of a velocities distribution interval. New sphere will have random velocity within the range velocity±velocityRange.

`volumeSection`(*=false, define factory by facets.*)
: Create new spheres inside factory volume rather than on its surface.

class `yade.wrapper.SpheresFactory`(*inherits GlobalEngine → Engine → Serializable*)

Engine for spitting spheres based on mass flow rate, particle size distribution etc. Initial velocity of particles is given by *vMin*, *vMax*, the *massFlowRate* determines how many particles to generate at each step. When *goalMass* is attained or positive *maxParticles* is reached, the engine does not produce particles anymore. Geometry of the region should be defined in a derived engine by overridden SpheresFactory::pickRandomPosition().

A sample script for this engine is in scripts/spheresFactory.py.

`PSDcalculateMass`(*=true*)
: PSD-Input is in mass (true), otherwise the number of particles will be considered.

`PSDcum`(*=uninitalized*)
: PSD-dispersion, cumulative procent meanings [-]

PSDsizes(*=uninitalized*)
: PSD-dispersion, sizes of cells, Diameter [m]

blockedDOFs(*=""*)
: Blocked degress of freedom

color(*=Vector3r(-1, -1, -1)*)
: Use the color for newly created particles, if specified

exactDiam(*=true*)
: If true, the particles only with the defined in PSDsizes diameters will be created. Otherwise the diameter will be randomly chosen in the range [PSDsizes[i-1]:PSDsizes[i]], in this case the length of PSDsizes should be more on 1, than the length of PSDcum.

goalMass(*=0*)
: Total mass that should be attained at the end of the current step. *(auto-updated)*

ids(*=uninitalized*)
: ids of created bodies

mask(*=-1*)
: groupMask to apply for newly created spheres

massFlowRate(*=NaN*)
: Mass flow rate [kg/s]

materialId(*=-1*)
: Shared material id to use for newly created spheres (can be negative to count from the end)

maxAttempt(*=5000*)
: Maximum number of attempts to position a new sphere randomly.

maxMass(*=-1*)
: Maximal mass at which to stop generating new particles regardless of massFlowRate. if maxMass=-1 - this parameter is ignored.

maxParticles(*=100*)
: The number of particles at which to stop generating new ones regardless of massFlowRate. if maxParticles=-1 - this parameter is ignored .

normal(*=Vector3r(NaN, NaN, NaN)*)
: Spitting direction (and orientation of the region's geometry).

numParticles(*=0*)
: Cummulative number of particles produces so far *(auto-updated)*

rMax(*=NaN*)
: Maximum radius of generated spheres (uniform distribution)

rMin(*=NaN*)
: Minimum radius of generated spheres (uniform distribution)

silent(*=false*)
: If true no complain about excessing maxAttempt but disable the factory (by set massFlowRate=0).

stopIfFailed(*=true*)
: If true, the SpheresFactory stops (sets massFlowRate=0), when maximal number of attempts to insert particle exceed.

totalMass(*=0*)
: Mass of spheres that was produced so far. *(auto-updated)*

totalVolume(*=0*)
: Volume of spheres that was produced so far. *(auto-updated)*

vAngle(*=NaN*)
: Maximum angle by which the initial sphere velocity deviates from the normal.

vMax(=*NaN*)
Maximum velocity norm of generated spheres (uniform distribution)

vMin(=*NaN*)
Minimum velocity norm of generated spheres (uniform distribution)

class yade.wrapper.**TesselationWrapper**(*inherits GlobalEngine → Engine → Serializable*)
Handle the triangulation of spheres in a scene, build tesselation on request, and give access to computed quantities : currently volume and porosity of each Voronoï sphere. Example script :

tt=TriaxialTest()

tt.generate('test.xml')

O.load('test.xml')

O.run(100)

TW=TesselationWrapper()

TW.triangulate() //compute regular Delaunay triangulation, don't construct tesselation

TW.computeVolumes() //will silently tesselate the packing

TW.volume(10) //get volume associated to sphere of id 10

ComputeDeformations() → None
Compute per-particle deformation. Get it with :yref:'TesselationWrapper.deformation'(id,i,j).

computeVolumes() → None
Compute volumes of all Voronoi's cells.

defToVtk([*(str)outputFile='def.vtk'*]) → None
Write local deformations in vtk format from states 0 and 1.

defToVtkFromPositions([*(str)input1='pos1'*[, *(str)input2='pos2'*[, *(str)outputFile='def.vtk'*[, *(bool)bz2=False*]]]]) → None
Write local deformations in vtk format from positions files (one sphere per line, with x,y,z,rad separated by spaces).

defToVtkFromStates([*(str)input1='state1'*[, *(str)input2='state2'*[, *(str)outputFile='def.vtk'*[, *(bool)bz2=False*]]]]) → None
Write local deformations in vtk format from state files (since the file format is very special, consider using defToVtkFromPositions if the input files were not generated by Tesselation-Wrapper).

deformation(*(int)id*, *(int)i*, *(int)j*) → float
Get particle deformation

getVolPoroDef([*(bool)deformation=False*]) → dict
Return a table with per-sphere computed quantities. Include deformations on the increment defined by states 0 and 1 if deformation=True (make sure to define states 0 and 1 consistently).

loadState(*(str)arg2*, *(bool)arg3*[, *(bool)stateNumber=0*]) → None
Load a file with positions to define state 0 or 1.

n_spheres(=*0*)
(auto-computed)

saveState([*(str)outputFile='state'*[, *(bool)state=0*[, *(bool)bz2=True*]]]) → None
Save a file with positions, can be later reloaded in order to define state 0 or 1.

setState([*(bool)state=0*]) → None
Make the current state of the simulation the initial (0) or final (1) configuration for the definition of displacement increments, use only state=0 if you just want to get volmumes and porosity.

triangulate([*(bool)reset=True*]) → None
triangulate spheres of the packing

volume([*(int)id=0*]) → float
Returns the volume of Voronoi's cell of a sphere.

class yade.wrapper.**TetraVolumetricLaw**(*inherits GlobalEngine → Engine → Serializable*)
Calculate physical response of 2 tetrahedra in interaction, based on penetration configuration given by TTetraGeom.

class yade.wrapper.**TimeStepper**(*inherits GlobalEngine → Engine → Serializable*)
Engine defining time-step (fundamental class)

active(*=true*)
is the engine active?

timeStepUpdateInterval(*=1*)
dt update interval

class yade.wrapper.**TorqueRecorder**(*inherits Recorder → PeriodicEngine → GlobalEngine → Engine → Serializable*)
Engine saves the total torque according to the given axis and ZeroPoint, the force is taken from bodies, listed in *ids* For instance, can be useful for defining the torque, which affects on ball mill during its work.

ids(*=uninitalized*)
List of bodies whose state will be measured

rotationAxis(*=Vector3r::UnitX()*)
Rotation axis

totalTorque(*=0*)
Resultant torque, returning by the function.

zeroPoint(*=Vector3r::Zero()*)
Point of rotation center

class yade.wrapper.**TriaxialStateRecorder**(*inherits Recorder → PeriodicEngine → GlobalEngine → Engine → Serializable*)
Engine recording triaxial variables (see the variables list in the first line of the output file). This recorder needs TriaxialCompressionEngine or ThreeDTriaxialEngine present in the simulation).

porosity(*=1*)
porosity of the packing [-]

class yade.wrapper.**VTKRecorder**(*inherits PeriodicEngine → GlobalEngine → Engine → Serializable*)
Engine recording snapshots of simulation into series of *.vtu files, readable by VTK-based post-processing programs such as Paraview. Both bodies (spheres and facets) and interactions can be recorded, with various vector/scalar quantities that are defined on them.

PeriodicEngine.initRun is initialized to `True` automatically.

ascii(*=false*)
Store data as readable text in the XML file (sets vtkXMLWriter data mode to `vtkXMLWriter::Ascii`, while the default is `Appended`

compress(*=false*)
Compress output XML files [experimental].

fileName(*="“*)
Base file name; it will be appended with {spheres,intrs,facets}-243100.vtu (unless *multiblock* is `True`) depending on active recorders and step number (243100 in this case). It can contain slashes, but the directory must exist already.

mask(*=0*)
If mask defined, only bodies with corresponding groupMask will be exported. If 0, all bodies will be exported.

recorders
List of active recorders (as strings). `all` (the default value) enables all base and generic recorders.

Base recorders

Base recorders save the geometry (unstructured grids) on which other data is defined. They are implicitly activated by many of the other recorders. Each of them creates a new file (or a block, if multiblock is set).

spheres Saves positions and radii (`radii`) of spherical particles.

facets Save facets positions (vertices).

intr Store interactions as lines between nodes at respective particles positions. Additionally stores magnitude of normal (`forceN`) and shear (`absForceT`) forces on interactions (the geom).

Generic recorders

Generic recorders do not depend on specific model being used and save commonly useful data.

id Saves id's (field `id`) of spheres; active only if `spheres` is active.

mass Saves masses (field `mass`) of spheres; active only if `spheres` is active.

clumpId Saves id's of clumps to which each sphere belongs (field `clumpId`); active only if `spheres` is active.

colors Saves colors of spheres and of facets (field `color`); only active if `spheres` or `facets` are activated.

mask Saves groupMasks of spheres and of facets (field `mask`); only active if `spheres` or `facets` are activated.

materialId Saves materialID of spheres and of facets; only active if `spheres` or `facets` are activated.

velocity Saves linear and angular velocities of spherical particles as Vector3 and length(fields `linVelVec`, `linVelLen` and `angVelVec`, `angVelLen` respectively``); only effective with `spheres`.

stress Saves stresses of spheres and of facets as Vector3 and length; only active if `spheres` or `facets` are activated.

Specific recorders

The following should only be activated in appropriate cases, otherwise crashes can occur due to violation of type presuppositions.

cpm Saves data pertaining to the concrete model: `cpmDamage` (normalized residual strength averaged on particle), `cpmStress` (stress on particle); `intr` is activated automatically by `cpm`

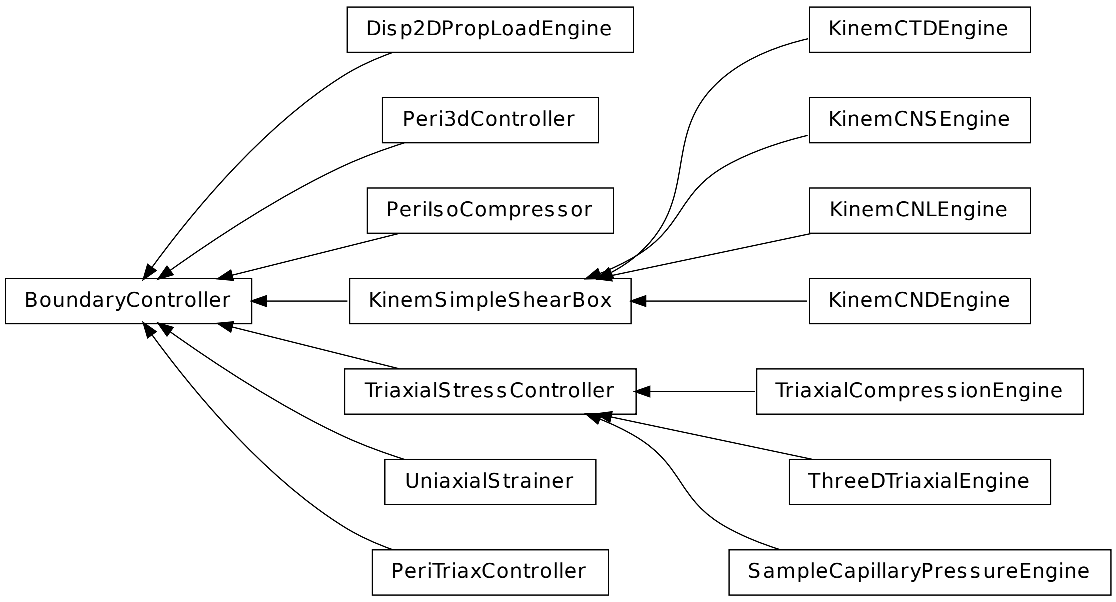
rpm Saves data pertaining to the rock particle model: `rpmSpecNum` shows different pieces of separated stones, only ids. `rpmSpecMass` shows masses of separated stones.

wpm Saves data pertaining to the wire particle model: `wpmForceNFactor` shows the loading factor for the wire, e.g. normal force divided by threshold normal force.

skipFacetIntr(=*true*)
Skip interactions with facets, when saving interactions

skipNondynamic(=*false*)
Skip non-dynamic spheres (but not facets).

7.3.2 BoundaryController



class **`yade.wrapper.BoundaryController`**(*inherits GlobalEngine → Engine → Serializable*)
Base for engines controlling boundary conditions of simulations. Not to be used directly.

class **`yade.wrapper.Disp2DPropLoadEngine`**(*inherits BoundaryController → GlobalEngine → Engine → Serializable*)
Disturbs a simple shear sample in a given displacement direction

This engine allows one to apply, on a simple shear sample, a loading controlled by du/dgamma = cste, which is equivalent to du + cste' * dgamma = 0 (proportionnal path loadings). To do so, the upper plate of the simple shear box is moved in a given direction (corresponding to a given du/dgamma), whereas lateral plates are moved so that the box remains closed. This engine can easily be used to perform directionnal probes, with a python script launching successivly the same .xml which contains this engine, after having modified the direction of loading (see *theta* attribute). That's why this Engine contains a *saveData* procedure which can save data on the state of the sample at the end of the loading (in case of successive loadings - for successive directions - through a python script, each line would correspond to one direction of loading).

`Key`(*="“*)
string to add at the names of the saved files, and of the output file filled by *saveData*

`LOG`(*=false*)
boolean controling the output of messages on the screen

`id_boxback`(*=4*)
the id of the wall at the back of the sample

`id_boxbas`(*=1*)
the id of the lower wall

`id_boxfront`(*=5*)
the id of the wall in front of the sample

`id_boxleft`(*=0*)
the id of the left wall

`id_boxright`(*=2*)
the id of the right wall

`id_topbox`(*=3*)
the id of the upper wall

nbre_iter(*=0*)
the number of iterations of loading to perform

theta(*=0.0*)
the angle, in a (gamma,h=-u) plane from the gamma - axis to the perturbation vector (trigo wise) [degrees]

v(*=0.0*)
the speed at which the perturbation is imposed. In case of samples which are more sensitive to normal loadings than tangential ones, one possibility is to take v = V_shear - | (V_shear-V_comp)*sin(theta) | => v=V_shear in shear; V_comp in compression [m/s]

class yade.wrapper.**KinemCNDEngine**(*inherits KinemSimpleShearBox → BoundaryController → GlobalEngine → Engine → Serializable*)
To apply a Constant Normal Displacement (CND) shear for a parallelogram box

This engine, designed for simulations implying a simple shear box (SimpleShear Preprocessor or scripts/simpleShear.py), allows one to perform a constant normal displacement shear, by translating horizontally the upper plate, while the lateral ones rotate so that they always keep contact with the lower and upper walls.

gamma(*=0.0*)
the current value of the tangential displacement

gamma_save(*=uninitalized*)
vector with the values of gamma at which a save of the simulation is performed [m]

gammalim(*=0.0*)
the value of the tangential displacement at wich the displacement is stopped [m]

shearSpeed(*=0.0*)
the speed at which the shear is performed : speed of the upper plate [m/s]

class yade.wrapper.**KinemCNLEngine**(*inherits KinemSimpleShearBox → BoundaryController → GlobalEngine → Engine → Serializable*)
To apply a constant normal stress shear (i.e. Constant Normal Load : CNL) for a parallelogram box (simple shear box : SimpleShear Preprocessor or scripts/simpleShear.py)

This engine allows one to translate horizontally the upper plate while the lateral ones rotate so that they always keep contact with the lower and upper walls.

In fact the upper plate can move not only horizontally but also vertically, so that the normal stress acting on it remains constant (this constant value is not chosen by the user but is the one that exists at the beginning of the simulation)

The right vertical displacements which will be allowed are computed from the rigidity Kn of the sample over the wall (so to cancel a deltaSigma, a normal dplt deltaSigma*S/(Kn) is set)

The movement is moreover controlled by the user via a *shearSpeed* which will be the speed of the upper wall, and by a maximum value of horizontal displacement *gammalim*, after which the shear stops.

Note: Not only the positions of walls are updated but also their speeds, which is all but useless considering the fact that in the contact laws these velocities of bodies are used to compute values of tangential relative displacements.

Warning: Because of this last point, if you want to use later saves of simulations executed with this Engine, but without that stopMovement was executed, your boxes will keep their speeds => you will have to cancel them 'by hand' in the .xml.

gamma(*=0.0*)
current value of tangential displacement [m]

gamma_save(*=uninitalized*)
vector with the values of gamma at which a save of the simulation is performed [m]

gammalim(*=0.0*)
: the value of tangential displacement (of upper plate) at wich the shearing is stopped [m]

shearSpeed(*=0.0*)
: the speed at wich the shearing is performed : speed of the upper plate [m/s]

class yade.wrapper.**KinemCNSEngine**(*inherits KinemSimpleShearBox → BoundaryController → GlobalEngine → Engine → Serializable*)

To apply a Constant Normal Stifness (CNS) shear for a parallelogram box (simple shear)

This engine, useable in simulations implying one deformable parallelepipedic box, allows one to translate horizontally the upper plate while the lateral ones rotate so that they always keep contact with the lower and upper walls. The upper plate can move not only horizontally but also vertically, so that the normal rigidity defined by DeltaF(upper plate)/DeltaU(upper plate) = constant (= *KnC* defined by the user).

The movement is moreover controlled by the user via a *shearSpeed* which is the horizontal speed of the upper wall, and by a maximum value of horizontal displacement *gammalim* (of the upper plate), after which the shear stops.

Note: not only the positions of walls are updated but also their speeds, which is all but useless considering the fact that in the contact laws these velocities of bodies are used to compute values of tangential relative displacements.

Warning: But, because of this last point, if you want to use later saves of simulations executed with this Engine, but without that stopMovement was executed, your boxes will keep their speeds => you will have to cancel them by hand in the .xml

KnC(*=10.0e6*)
: the normal rigidity chosen by the user [MPa/mm] - the conversion in Pa/m will be made

gamma(*=0.0*)
: current value of tangential displacement [m]

gammalim(*=0.0*)
: the value of tangential displacement (of upper plate) at wich the shearing is stopped [m]

shearSpeed(*=0.0*)
: the speed at wich the shearing is performed : speed of the upper plate [m/s]

class yade.wrapper.**KinemCTDEngine**(*inherits KinemSimpleShearBox → BoundaryController → GlobalEngine → Engine → Serializable*)

To compress a simple shear sample by moving the upper box in a vertical way only, so that the tangential displacement (defined by the horizontal gap between the upper and lower boxes) remains constant (thus, the CTD = Constant Tangential Displacement). The lateral boxes move also to keep always contact. All that until this box is submitted to a given stress (=*targetSigma*). Moreover saves are executed at each value of stresses stored in the vector *sigma_save*, and at *targetSigma*

compSpeed(*=0.0*)
: (vertical) speed of the upper box : >0 for real compression, <0 for unloading [m/s]

sigma_save(*=uninitalized*)
: vector with the values of sigma at which a save of the simulation should be performed [kPa]

targetSigma(*=0.0*)
: the value of sigma at which the compression should stop [kPa]

class yade.wrapper.**KinemSimpleShearBox**(*inherits BoundaryController → GlobalEngine → Engine → Serializable*)

This class is supposed to be a mother class for all Engines performing loadings on the simple shear box of SimpleShear. It is not intended to be used by itself, but its declaration and implentation will thus contain all what is useful for all these Engines. The script simpleShear.py illustrates the use of the various corresponding Engines.

Key(=”“)
string to add at the names of the saved files

LOG(=*false*)
boolean controling the output of messages on the screen

alpha(=*Mathr::PI/2.0*)
the angle from the lower box to the left box (trigo wise). Measured by this Engine. Has to be saved, but not to be changed by the user.

f0(=*0.0*)
the (vertical) force acting on the upper plate on the very first time step (determined by the Engine). Controls of the loadings in case of KinemCNSEngine or KinemCNLEngine will be done according to this initial value [N]. Has to be saved, but not to be changed by the user.

firstRun(=*true*)
boolean set to false as soon as the engine has done its job one time : useful to know if initial height of, and normal force sustained by, the upper box are known or not (and thus if they have to be initialized). Has to be saved, but not to be changed by the user.

id_boxback(=*4*)
the id of the wall at the back of the sample

id_boxbas(=*1*)
the id of the lower wall

id_boxfront(=*5*)
the id of the wall in front of the sample

id_boxleft(=*0*)
the id of the left wall

id_boxright(=*2*)
the id of the right wall

id_topbox(=*3*)
the id of the upper wall

max_vel(=*1.0*)
to limit the speed of the vertical displacements done to control σ (CNL or CNS cases) [m/s]

temoin_save(=*uninitalized*)
vector (same length as ‘gamma_save’ for ex), with 0 or 1 depending whether the save for the corresponding value of gamma has been done (1) or not (0). Has to be saved, but not to be changed by the user.

wallDamping(=*0.2*)
the vertical displacements done to to control σ (CNL or CNS cases) are in fact damped, through this wallDamping

y0(=*0.0*)
the height of the upper plate at the very first time step : the engine finds its value [m]. Has to be saved, but not to be changed by the user.

class yade.wrapper.**Peri3dController**(*inherits BoundaryController → GlobalEngine → Engine → Serializable*)

Class for controlling independently all 6 components of “engineering” stress and strain of periodic :yref:‘‘Cell‘. goal are the goal values, while stressMask determines which components prescribe stress and which prescribe strain.

If the strain is prescribed, appropeiate strain rate is directly applied. If the stress is prescribed, the strain predictor is used: from stress values in two previous steps the value of strain rate is prescribed so as the value of stress in the next step is as close as possible to the ideal one. Current algorithm is extremly simple and probably will be changed in future, but is roboust enough and mostly works fine.

Stress error (difference between actual and ideal stress) is evaluated in current and previous steps ($d\sigma_i, d\sigma_{i-1}$. Linear extrapolation is used to estimate error in the next step

$$d\sigma_{i+1} = 2d\sigma_i - d\sigma_{i-1}$$

According to this error, the strain rate is modified by mod parameter

$$d\sigma_{i+1} \begin{cases} > 0 \rightarrow \dot{\varepsilon}_{i+1} = \dot{\varepsilon}_i - \max(\text{abs}(\dot{\boldsymbol{\varepsilon}}_i)) \cdot \text{mod} \\ < 0 \rightarrow \dot{\varepsilon}_{i+1} = \dot{\varepsilon}_i + \max(\text{abs}(\dot{\boldsymbol{\varepsilon}}_i)) \cdot \text{mod} \end{cases}$$

According to this fact, the prescribed stress will (almost) never have exact prescribed value, but the difference would be very small (and decreasing for increasing nSteps. This approach works good if one of the dominant strain rates is prescribed. If all stresses are prescribed or if all goal strains is prescribed as zero, a good estimation is needed for the first step, therefore the compliance matrix is estimated (from user defined estimations of macroscopic material parameters youngEstimation and poissonEstimation) and respective strain rates is computed form prescribed stress rates and compliance matrix (the estimation of compliance matrix could be computed autamatically avoiding user inputs of this kind).

The simulation on rotated periodic cell is also supported. Firstly, the polar decomposition is performed on cell's transformation matrix trsf $\mathcal{T} = \mathbf{UP}$, where $\mathbf{U}$ is orthogonal (unitary) matrix representing rotation and $\mathbf{P}$ is a positive semi-definite Hermitian matrix representing strain. A logarithm of $\mathbf{P}$ should be used to obtain realistic values at higher strain values (not implemented yet). A prescribed strain increment in global coordinates $dt \cdot \dot{\boldsymbol{\varepsilon}}$ is properly rotated to cell's local coordinates and added to $\mathbf{P}$

$$\mathbf{P}_{i+1} = \mathbf{P} + \mathbf{U}^T dt \cdot \dot{\boldsymbol{\varepsilon}}\mathbf{U}$$

The new value of trsf is computed at $\mathbf{T}_{i+1} = \mathbf{UP}_{i+1}$. From current and next trsf the cell's velocity gradient velGrad is computed (according to its definition) as

$$\mathbf{V} = (\mathbf{T}_{i+1}\mathbf{T}^{-1} - \mathbf{I})/dt$$

Current implementation allow user to define independent loading "path" for each prescribed component. i.e. define the prescribed value as a function of time (or progress or steps). See Paths.

Examples scripts/test/peri3dController_example1 and scripts/test/peri3dController_triaxialCompression explain usage and inputs of Peri3dController, scripts/test/peri3dController_shear is an example of using shear components and also simulation on rotatd cell.

`doneHook`(*=uninitalized*)
: Python command (as string) to run when nSteps is achieved. If empty, the engine will be set dead.

`goal`(*=Vector6r::Zero()*)
: Goal state; only the upper triangular matrix is considered; each component is either prescribed stress or strain, depending on stressMask.

`maxStrain`(*=1e6*)
: Maximal asolute value of strain allowed in the simulation. If reached, the simulation is considered as finished

`maxStrainRate`(*=1e3*)
: Maximal absolute value of strain rate (both normal and shear components of strain)

`mod`(*=.1*)
: Predictor modificator, by trail-and-error analysis the value 0.1 was found as the best.

`nSteps`(*=1000*)
: Number of steps of the simulation.

poissonEstimation(*=.25*)
Estimation of macroscopic Poisson's ratio, used used for the first simulation step

progress(*=0.*)
Actual progress of the simulation with Controller.

strain(*=Vector6r::Zero()*)
Current strain (deformation) vector (ε_x,ε_y,ε_z,γ_{yz},γ_{zx},γ_{xy}) *(auto-updated)*.

strainRate(*=Vector6r::Zero()*)
Current strain rate vector.

stress(*=Vector6r::Zero()*)
Current stress vector (σ_x,σ_y,σ_z,τ_{yz},τ_{zx},τ_{xy})|yupdate|.

stressIdeal(*=Vector6r::Zero()*)
Ideal stress vector at current time step.

stressMask(*=0, all strains*)
mask determining whether components of goal are strain (0) or stress (1). The order is 00,11,22,12,02,01 from the least significant bit. (e.g. 0b000011 is stress 00 and stress 11).

stressRate(*=Vector6r::Zero()*)
Current stress rate vector (that is prescribed, the actual one slightly differ).

xxPath
"Time function" (piecewise linear) for xx direction. Sequence of couples of numbers. First number is time, second number desired value of respective quantity (stress or strain). The last couple is considered as final state (equal to (nSteps, goal)), other values are relative to this state.

Example: nSteps=1000, goal[0]=300, xxPath=((2,3),(4,1),(5,2))

at step 400 (=5*1000/2) the value is 450 (=3*300/2),

at step 800 (=4*1000/5) the value is 150 (=1*300/2),

at step 1000 (=5*1000/5=nSteps) the value is 300 (=2*300/2=goal[0]).

See example scripts/test/peri3dController_example1 for illusration.

xyPath(*=vector<Vector2r>(1, Vector2r::Ones())*)
Time function for xy direction, see xxPath

youngEstimation(*=1e20*)
Estimation of macroscopic Young's modulus, used for the first simulation step

yyPath(*=vector<Vector2r>(1, Vector2r::Ones())*)
Time function for yy direction, see xxPath

yzPath(*=vector<Vector2r>(1, Vector2r::Ones())*)
Time function for yz direction, see xxPath

zxPath(*=vector<Vector2r>(1, Vector2r::Ones())*)
Time function for zx direction, see xxPath

zzPath(*=vector<Vector2r>(1, Vector2r::Ones())*)
Time function for zz direction, see xxPath

class yade.wrapper.**PeriIsoCompressor**(*inherits BoundaryController → GlobalEngine → Engine → Serializable*)
Compress/decompress cloud of spheres by controlling periodic cell size until it reaches prescribed average stress, then moving to next stress value in given stress series.

charLen(*=-1.*)
Characteristic length, should be something like mean particle diameter (default -1=invalid value))

currUnbalanced
Current value of unbalanced force

doneHook(*=”“*)
: Python command to be run when reaching the last specified stress

globalUpdateInt(*=20*)
: how often to recompute average stress, stiffness and unbalanced force

keepProportions(*=true*)
: Exactly keep proportions of the cell (stress is controlled based on average, not its components

maxSpan(*=-1.*)
: Maximum body span in terms of bbox, to prevent periodic cell getting too small. *(auto-computed)*

maxUnbalanced(*=1e-4*)
: if actual unbalanced force is smaller than this number, the packing is considered stable,

sigma
: Current stress value

state(*=0*)
: Where are we at in the stress series

stresses(*=uninitalized*)
: Stresses that should be reached, one after another

class `yade.wrapper.PeriTriaxController`(*inherits BoundaryController → GlobalEngine → Engine → Serializable*)
: Engine for independently controlling stress or strain in periodic simulations.

`strainStress` contains absolute values for the controlled quantity, and `stressMask` determines meaning of those values (0 for strain, 1 for stress): e.g. `( 1<<0 | 1<<2 ) = 1 | 4 = 5` means that `strainStress[0]` and `strainStress[2]` are stress values, and `strainStress[1]` is strain.

See scripts/test/periodic-triax.py for a simple example.

absStressTol(*=1e3*)
: Absolute stress tolerance

currUnbalanced(*=NaN*)
: current unbalanced force (updated every globUpdate) *(auto-updated)*

doneHook(*=uninitalized*)
: python command to be run when the desired state is reached

dynCell(*=false*)
: Imposed stress can be controlled using the packing stiffness or by applying the laws of dynamic (dynCell=true). Don’t forget to assign a mass to the cell.

externalWork(*=0*)
: Work input from boundary controller.

globUpdate(*=5*)
: How often to recompute average stress, stiffness and unbalaced force.

goal
: Desired stress or strain values (depending on stressMask), strains defined as `strain(i)=log(Fii)`.

Warning: Strains are relative to the O.cell.refSize (reference cell size), not the current one (e.g. at the moment when the new strain value is set).

growDamping(*=.25*)
: Damping of cell resizing (0=perfect control, 1=no control at all); see also `wallDamping` in TriaxialStressController.

mass(*=NaN*)
: mass of the cell (user set); if not set and dynCell is used, it will be computed as sum of masses of all particles.

maxBodySpan(*=Vector3r::Zero()*)
: maximum body dimension *(auto-computed)*

maxStrainRate(*=Vector3r(1, 1, 1)*)
: Maximum strain rate of the periodic cell.

maxUnbalanced(*=1e-4*)
: maximum unbalanced force.

prevGrow(*=Vector3r::Zero()*)
: previous cell grow

relStressTol(*=3e-5*)
: Relative stress tolerance

stiff(*=Vector3r::Zero()*)
: average stiffness (only every globUpdate steps recomputed from interactions) *(auto-updated)*

strain(*=Vector3r::Zero()*)
: cell strain *(auto-updated)*

strainRate(*=Vector3r::Zero()*)
: cell strain rate *(auto-updated)*

stress(*=Vector3r::Zero()*)
: diagonal terms of the stress tensor

stressMask(*=0, all strains*)
: mask determining strain/stress (0/1) meaning for goal components

stressTensor(*=Matrix3r::Zero()*)
: average stresses, updated at every step (only every globUpdate steps recomputed from interactions if !dynCell)

useDem3Dof(*=false*)
: For some constitutive laws (practicaly all laws based on Dem3Dof), normalForce and shearForce on interactions are in the reverse sense and this flag must be true. See this message.

class yade.wrapper.**SampleCapillaryPressureEngine**(*inherits TriaxialStressController → BoundaryController → GlobalEngine → Engine → Serializable*)

It produces the isotropic compaction of an assembly and allows one to controlled the capillary pressure inside (uses Law2_ScGeom_CapillaryPhys_Capillarity).

Pressure(*=0*)
: Value of the capillary pressure Uc=Ugas-Uliquid (see Law2_ScGeom_CapillaryPhys_Capillarity). [Pa]

PressureVariation(*=0*)
: Variation of the capillary pressure (each iteration). [Pa]

SigmaPrecision(*=0.001*)
: tolerance in terms of mean stress to consider the packing as stable

StabilityCriterion(*=0.01*)
: tolerance in terms of :yref:'TriaxialCompressionEngine::UnbalancedForce' to consider the packing as stable

UnbalancedForce(*=1*)
: mean resultant forces divided by mean contact force

binaryFusion(*=1*)
: If yes, capillary force are set to 0 when, at least, 1 overlap is detected for a meniscus. If no, capillary force is divided by the number of overlaps.

fusionDetection(*=1*)
: Is the detection of menisci overlapping activated?

pressureVariationActivated(*=1*)
: Is the capillary pressure varying?

class `yade.wrapper.ThreeDTriaxialEngine`(*inherits TriaxialStressController → BoundaryController → GlobalEngine → Engine → Serializable*)

The engine perform a triaxial compression with a control in direction 'i' in stress (if stressControl_i) else in strain.

For a stress control the imposed stress is specified by 'sigma_i' with a 'max_veli' depending on 'strainRatei'. To obtain the same strain rate in stress control than in strain control you need to set 'wallDamping = 0.8'. For a strain control the imposed strain is specified by 'strainRatei'. With this engine you can also perform internal compaction by growing the size of particles by using `TriaxialStressController::controlInternalStress`. For that, just switch on 'internalCompaction=1' and fix sigma_iso=value of mean pressure that you want at the end of the internal compaction.

`Key`(*="“*)
A string appended at the end of all files, use it to name simulations.

`UnbalancedForce`(*=1*)
mean resultant forces divided by mean contact force

`currentStrainRate1`(*=0*)
current strain rate in direction 1 - converging to :yref:'ThreeDTriaxialEngine::strainRate1' (./s)

`currentStrainRate2`(*=0*)
current strain rate in direction 2 - converging to :yref:'ThreeDTriaxialEngine::strainRate2' (./s)

`currentStrainRate3`(*=0*)
current strain rate in direction 3 - converging to :yref:'ThreeDTriaxialEngine::strainRate3' (./s)

`frictionAngleDegree`(*=-1*)
Value of friction used in the simulation if (updateFrictionAngle)

`setContactProperties`(*(float)arg2*) → None
Assign a new friction angle (degrees) to dynamic bodies and relative interactions

`strainDamping`(*=0.9997*)
factor used for smoothing changes in effective strain rate. If target rate is TR, then (1-damping)*(TR-currentRate) will be added at each iteration. With damping=0, rate=target all the time. With damping=1, it doesn't change.

`strainRate1`(*=0*)
target strain rate in direction 1 (./s)

`strainRate2`(*=0*)
target strain rate in direction 2 (./s)

`strainRate3`(*=0*)
target strain rate in direction 3 (./s)

`stressControl_1`(*=true*)
Switch to choose a stress or a strain control in directions 1

`stressControl_2`(*=true*)
Switch to choose a stress or a strain control in directions 2

`stressControl_3`(*=true*)
Switch to choose a stress or a strain control in directions 3

`updateFrictionAngle`(*=false*)
Switch to activate the update of the intergranular frictionto the value :yref:'ThreeDTriaxialEngine::frictionAngleDegree

class `yade.wrapper.TriaxialCompressionEngine`(*inherits TriaxialStressController → BoundaryController → GlobalEngine → Engine → Serializable*)

The engine is a state machine with the following states; transitions my be automatic, see below.

1. STATE_ISO_COMPACTION: isotropic compaction (compression) until the prescribed mean pressue sigmaIsoCompaction is reached and the packing is stable. The compaction happens either by straining the walls (!internalCompaction) or by growing size of grains (internalCompaction).
2. STATE_ISO_UNLOADING: isotropic unloading from the previously reached state, until the mean pressure sigmaLateralConfinement is reached (and stabilizes).

   Note: this state will be skipped if sigmaLateralConfinement == sigmaIsoCompaction.

3. STATE_TRIAX_LOADING: confined uniaxial compression: constant sigmaLateralConfinement is kept at lateral walls (left, right, front, back), while top and bottom walls load the packing in their axis (by straining), until the value of epsilonMax (deformation along the loading axis) is reached. At this point, the simulation is stopped.
4. STATE_FIXED_POROSITY_COMPACTION: isotropic compaction (compression) until a chosen porosity value (parameter:fixedPorosity). The six walls move with a chosen translation speed (parameter StrainRate).
5. STATE_TRIAX_LIMBO: currently unused, since simulation is hard-stopped in the previous state.

Transition from COMPACTION to UNLOADING is done automatically if autoUnload==true;

Transition from (UNLOADING to LOADING) or from (COMPACTION to LOADING: if UNLOADING is skipped) is done automatically if autoCompressionActivation=true; Both autoUnload and autoCompressionActivation are true by default.

Note: Most of the algorithms used have been developed initialy for simulations reported in [Chareyre2002a] and [Chareyre2005]. They have been ported to Yade in a second step and used in e.g. [Kozicki2008],[Scholtes2009b]_,[Jerier2010b].

`Key`(=”“)
: A string appended at the end of all files, use it to name simulations.

`StabilityCriterion`(=*0.001*)
: tolerance in terms of TriaxialCompressionEngine::UnbalancedForce to consider the packing is stable

`UnbalancedForce`(=*1*)
: mean resultant forces divided by mean contact force

`autoCompressionActivation`(=*true*)
: Auto-switch from isotropic compaction (or unloading state if sigmaLateralConfinement<sigmaIsoCompaction) to deviatoric loading

`autoStopSimulation`(=*true*)
: Stop the simulation when the sample reach STATE_LIMBO, or keep running

`autoUnload`(=*true*)
: Auto-switch from isotropic compaction to unloading

`currentState`(=*1*)
: There are 5 possible states in which TriaxialCompressionEngine can be. See above wrapper.TriaxialCompressionEngine

`currentStrainRate`(=*0*)
: current strain rate - converging to TriaxialCompressionEngine::strainRate (./s)

`epsilonMax`(=*0.5*)
: Value of axial deformation for which the loading must stop

fixedPoroCompaction(*=false*)
: A special type of compaction with imposed final porosity TriaxialCompressionEngine::fixedPorosity (WARNING : can give unrealistic results!)

fixedPorosity(*=0*)
: Value of porosity chosen by the user

frictionAngleDegree(*=-1*)
: Value of friction assigned just before the deviatoric loading

maxStress(*=0*)
: Max value of stress during the simulation (for post-processing)

noFiles(*=false*)
: If true, no files will be generated (*.xml, *.spheres,...)

previousSigmaIso(*=1*)
: Previous value of inherited sigma_iso (used to detect manual changes of the confining pressure)

previousState(*=1*)
: Previous state (used to detect manual changes of the state in .xml)

setContactProperties(*(float)arg2*) → None
: Assign a new friction angle (degrees) to dynamic bodies and relative interactions

sigmaIsoCompaction(*=1*)
: Prescribed isotropic pressure during the compaction phase

sigmaLateralConfinement(*=1*)
: Prescribed confining pressure in the deviatoric loading; might be different from TriaxialCompressionEngine::sigmaIsoCompaction

strainRate(*=0*)
: target strain rate (./s)

testEquilibriumInterval(*=20*)
: interval of checks for transition between phases, higher than 1 saves computation time.

translationAxis(*=TriaxialStressController::normal[wall_bottom]*)
: compression axis

uniaxialEpsilonCurr(*=1*)
: Current value of axial deformation during confined loading (is reference to strain[1])

class yade.wrapper.**TriaxialStressController**(*inherits BoundaryController → GlobalEngine → Engine → Serializable*)

An engine maintaining constant stresses on some boundaries of a parallepipedic packing. See also TriaxialCompressionEngine

Note: The algorithms used have been developed initialy for simulations reported in [Chareyre2002a] and [Chareyre2005]. They have been ported to Yade in a second step and used in e.g. [Kozicki2008],[Scholtes2009b]_,[Jerier2010b].

boxVolume
: Total packing volume.

computeStressStrainInterval(*=10*)

depth(*=0*)
: size of the box (2-axis) *(auto-updated)*

depth0(*=0*)
: Reference size for strain definition. See TriaxialStressController::depth

externalWork(*=0*)
: Energy provided by boundaries.|yupdate|

finalMaxMultiplier(*=1.00001*)
: max multiplier of diameters during internal compaction (secondary precise adjustment - TriaxialStressController::maxMultiplier is used in the initial stage)

height(*=0*)
: size of the box (1-axis) *(auto-updated)*

height0(*=0*)
: Reference size for strain definition. See TriaxialStressController::height

internalCompaction(*=true*)
: Switch between 'external' (walls) and 'internal' (growth of particles) compaction.

isAxisymetric(*=true*)
: if true, sigma_iso is assigned to sigma1, 2 and 3 (applies at each iteration and overrides user-set values of s1,2,3)

maxMultiplier(*=1.001*)
: max multiplier of diameters during internal compaction (initial fast increase - TriaxialStressController::finalMaxMultiplier is used in a second stage)

max_vel(*=0.001*)
: Maximum allowed walls velocity [m/s]. This value superseeds the one assigned by the stress controller if the later is higher. max_vel can be set to infinity in many cases, but sometimes helps stabilizing packings. Based on this value, different maxima are computed for each axis based on the dimensions of the sample, so that if each boundary moves at its maximum velocity, the strain rate will be isotropic (see e.g. TriaxialStressController::max_vel1).

max_vel1
: see TriaxialStressController::max_vel *(auto-computed)*

max_vel2
: see TriaxialStressController::max_vel *(auto-computed)*

max_vel3
: see TriaxialStressController::max_vel *(auto-computed)*

meanStress(*=0*)
: Mean stress in the packing. *(auto-updated)*

porosity
: Porosity of the packing.

previousMultiplier(*=1*)
: *(auto-updated)*

previousStress(*=0*)
: *(auto-updated)*

radiusControlInterval(*=10*)

sigma1(*=0*)
: prescribed stress on axis 1 (see TriaxialStressController::isAxisymetric)

sigma2(*=0*)
: prescribed stress on axis 2 (see TriaxialStressController::isAxisymetric)

sigma3(*=0*)
: prescribed stress on axis 3 (see TriaxialStressController::isAxisymetric)

sigma_iso(*=0*)
: prescribed confining stress (see TriaxialStressController::isAxisymetric)

spheresVolume
: Total volume pf spheres.

stiffnessUpdateInterval(*=10*)
: target strain rate (./s)

strain
Current strain in a vector (exx,eyy,ezz). The values reflect true (logarithmic) strain.

stress(*(int)id*) → Vector3
Return the mean stress vector acting on boundary 'id', with 'id' between 0 and 5.

thickness(*=-1*)
thickness of boxes (needed by some functions)

volumetricStrain(*=0*)
Volumetric strain (see TriaxialStressController::strain).|yupdate|

wallDamping(*=0.25*)
wallDamping coefficient - wallDamping=0 implies a (theoretical) perfect control, wallDamping=1 means no movement

wall_back_activated(*=true*)
if true, the engine is keeping stress constant on this boundary.

wall_back_id(*=4*)
id of boundary ; coordinate 2- (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_bottom_activated(*=true*)
if true, the engine is keeping stress constant on this boundary.

wall_bottom_id(*=2*)
id of boundary ; coordinate 1- (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_front_activated(*=true*)
if true, the engine is keeping stress constant on this boundary.

wall_front_id(*=5*)
id of boundary ; coordinate 2+ (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_left_activated(*=true*)
if true, the engine is keeping stress constant on this boundary.

wall_left_id(*=0*)
id of boundary ; coordinate 0- (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_right_activated(*=true*)
if true, the engine is keeping stress constant on this boundary.

wall_right_id(*=1*)
id of boundary ; coordinate 0+ (default value is ok if aabbWalls are appended BEFORE spheres.)

wall_top_activated(*=true*)
if true, the engine is keeping stress constant on this boundary.

wall_top_id(*=3*)
id of boundary ; coordinate 1+ (default value is ok if aabbWalls are appended BEFORE spheres.)

width(*=0*)
size of the box (0-axis) *(auto-updated)*

width0(*=0*)
Reference size for strain definition. See TriaxialStressController::width

class yade.wrapper.**UniaxialStrainer**(*inherits BoundaryController → GlobalEngine → Engine → Serializable*)
Axial displacing two groups of bodies in the opposite direction with given strain rate.

absSpeed(*=NaN*)
alternatively, absolute speed of boundary motion can be specified; this is effective only at the

beginning and if strainRate is not set; changing absSpeed directly during simulation wil have no effect. [ms ^{1}]

active(*=true*)
Whether this engine is activated

asymmetry(*=0, symmetric*)
If 0, straining is symmetric for negIds and posIds; for 1 (or -1), only posIds are strained and negIds don't move (or vice versa)

avgStress(*=0*)
Current average stress *(auto-updated)* [Pa]

axis(*=2*)
The axis which is strained (0,1,2 for x,y,z)

blockDisplacements(*=false*)
Whether displacement of boundary bodies perpendicular to the strained axis are blocked of are free

blockRotations(*=false*)
Whether rotations of boundary bodies are blocked.

crossSectionArea(*=NaN*)
crossSection perpendicular to he strained axis; must be given explicitly [m^{2}]

currentStrainRate(*=NaN*)
Current strain rate (update automatically). *(auto-updated)*

idleIterations(*=0*)
Number of iterations that will pass without straining activity after stopStrain has been reached

initAccelTime(*=-200*)
Time for strain reaching the requested value (linear interpolation). If negative, the time is dt*(-initAccelTime), where dt is the timestep at the first iteration. [s]

limitStrain(*=0, disabled*)
Invert the sense of straining (sharply, without transition) one this value of strain is reached. Not effective if 0.

negIds(*=uninitalized*)
Bodies on which strain will be applied (on the negative end along the axis)

notYetReversed(*=true*)
Flag whether the sense of straining has already been reversed (only used internally).

originalLength(*=NaN*)
Distance of reference bodies in the direction of axis before straining started (computed automatically) [m]

posIds(*=uninitalized*)
Bodies on which strain will be applied (on the positive end along the axis)

setSpeeds(*=false*)
should we set speeds at the beginning directly, instead of increasing strain rate progressively?

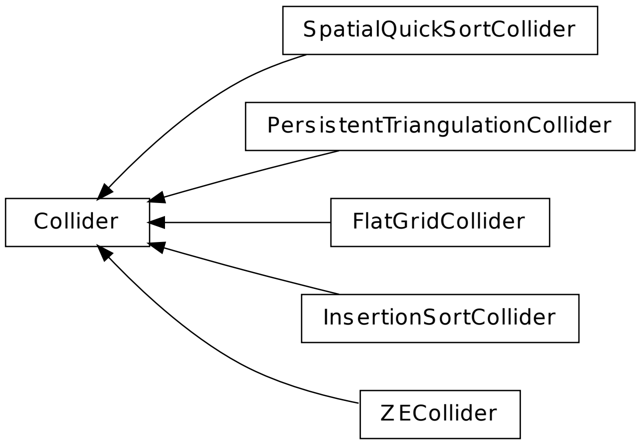
stopStrain(*=NaN*)
Strain at which we will pause simulation; inactive (nan) by default; must be reached from below (in absolute value)

strain(*=0*)
Current strain value, elongation/originalLength *(auto-updated)* [-]

strainRate(*=NaN*)
Rate of strain, starting at 0, linearly raising to strainRate. [-]

stressUpdateInterval(*=10*)
How often to recompute stress on supports.

7.3.3 Collider



class `yade.wrapper.Collider`(*inherits GlobalEngine → Engine → Serializable*)

Abstract class for finding spatial collisions between bodies.

Special constructor

Derived colliders (unless they override `pyHandleCustomCtorArgs`) can be given list of BoundFunctors which is used to initialize the internal boundDispatcher instance.

`boundDispatcher`(*=new BoundDispatcher*)

BoundDispatcher object that is used for creating bounds on collider's request as necessary.

class `yade.wrapper.FlatGridCollider`(*inherits Collider → GlobalEngine → Engine → Serializable*)

Non-optimized grid collider, storing grid as dense flat array. Each body is assigned to (possibly multiple) cells, which are arranged in regular grid between *aabbMin* and *aabbMax*, with cell size *step* (same in all directions). Bodies outsize (*aabbMin*, *aabbMax*) are handled gracefully, assigned to closest cells (this will create spurious potential interactions). *verletDist* determines how much is each body enlarged to avoid collision detection at every step.

Note: This collider keeps all cells in linear memory array, therefore will be memory-inefficient for sparse simulations.

Note: Periodic boundary is not handled (yet).

`aabbMax`(*=Vector3r::Zero()*)

Upper corner of grid (approximate, might be rouded up to *minStep*.

`aabbMin`(*=Vector3r::Zero()*)

Lower corner of grid.

`step`(*=0*)

Step in the grid (cell size)

`verletDist`(*=0*)

Length by which enlarge space occupied by each particle; avoids running collision detection at every step.

class `yade.wrapper.InsertionSortCollider`(*inherits Collider → GlobalEngine → Engine → Serializable*)

Collider with O(n log(n)) complexity, using Aabb for bounds.

At the initial step, Bodies' bounds (along sortAxis) are first std::sort'ed along one axis (sortAxis), then collided. The initial sort has $O(n^2)$ complexity, see Colliders' performance for some information (There are scripts in examples/collider-perf for measurements).

Insertion sort is used for sorting the bound list that is already pre-sorted from last iteration, where each inversion calls checkOverlap which then handles either overlap (by creating interaction if necessary) or its absence (by deleting interaction if it is only potential).

Bodies without bounding volume (such as clumps) are handled gracefully and never collide. Deleted bodies are handled gracefully as well.

This collider handles periodic boundary conditions. There are some limitations, notably:

1. No body can have Aabb larger than cell's half size in that respective dimension. You get exception if it does and gets in interaction. One way to explicitly by-pass this restriction is offered by `allowBiggerThanPeriod`, which can be turned on to insert a floor in the form of a very large box for instance (see examples/periodicSandPile.py).
2. No body can travel more than cell's distance in one step; this would mean that the simulation is numerically exploding, and it is only detected in some cases.

Stride can be used to avoid running collider at every step by enlarging the particle's bounds, tracking their displacements and only re-run if they might have gone out of that bounds (see Verlet list for brief description and background) . This requires cooperation from NewtonIntegrator as well as BoundDispatcher, which will be found among engines automatically (exception is thrown if they are not found).

If you wish to use strides, set `verletDist` (length by which bounds will be enlarged in all directions) to some value, e.g. 0.05 × typical particle radius. This parameter expresses the tradeoff between many potential interactions (running collider rarely, but with longer exact interaction resolution phase) and few potential interactions (running collider more frequently, but with less exact resolutions of interactions); it depends mainly on packing density and particle radius distribution.

If `targetInterv` is >1, not all particles will have their bound enlarged by `verletDist`; instead, they will have bounds increased by a length in order to trigger a new colliding after `targetInterv` iteration, assuming they move at almost constant velocity. Ideally in this method, all particles would reach their bounds at the sime iteration. This is of course not the case as soon as velocities fluctuate in time. Bound::sweepLength is tuned on the basis of the displacement recorded between the last two runs of the collider. In this situation, `verletDist` defines the maximum sweep length.

`allowBiggerThanPeriod`
If true, tests on bodies sizes will be disabled, and the simulation will run normaly even if bodies larger than period are found. It can be useful when the periodic problem include e.g. a floor modelized with wall/box/facet. Be sure you know what you are doing if you touch this flag. The result is undefined if one large body moves out of the (0,0,0) period.

`dumpBounds()` → tuple
Return representation of the internal sort data. The format is `([...],[...],[...])` for 3 axes, where each `...` is a list of entries (bounds). The entry is a tuple with the flowing items:

- coordinate (float)
- body id (int), but negated for negative bounds
- period numer (int), if the collider is in the periodic regime.

`fastestBodyMaxDist`(*=-1*)
Normalized maximum displacement of the fastest body since last run; if >= 1, we could get out of bboxes and will trigger full run. *(auto-updated)*

`minSweepDistFactor`(*=0.1*)
Minimal distance by which enlarge all bounding boxes; superseeds computed value of verletDist when lower that (minSweepDistFactor x verletDist).

`numReinit`(*=0*)
Cummulative number of bound array re-initialization.

periodic
Whether the collider is in periodic mode (read-only; for debugging) *(auto-updated)*

sortAxis(*=0*)
Axis for the initial contact detection.

sortThenCollide(*=false*)
Separate sorting and colliding phase; it is MUCH slower, but all interactions are processed at every step; this effectively makes the collider non-persistent, not remembering last state. (The default behavior relies on the fact that inversions during insertion sort are overlaps of bounding boxes that just started/ceased to exist, and only processes those; this makes the collider much more efficient.)

strideActive
Whether striding is active (read-only; for debugging). *(auto-updated)*

targetInterv(*=50*)
(experimental) Target number of iterations between bound update, used to define a smaller sweep distance for slower grains if >0, else always use 1*verletDist. Useful in simulations with strong velocity contrasts between slow bodies and fast bodies.

updatingDispFactor(*=-1*)
(experimental) Displacement factor used to trigger bound update: the bound is updated only if updatingDispFactor*disp>sweepDist when >0, else all bounds are updated.

useless(*=uninitalized*)
for compatibility of scripts defining the old collider's attributes - see deprecated attributes

verletDist(*=-.5, Automatically initialized*)
Length by which to enlarge particle bounds, to avoid running collider at every step. Stride disabled if zero. Negative value will trigger automatic computation, so that the real value will be |verletDist| × minimum spherical particle radius; if there are no spherical particles, it will be disabled. The actual length added to one bound can be only a fraction of verletDist when InsertionSortCollider::targetInterv is >0.

class yade.wrapper.**PersistentTriangulationCollider**(*inherits Collider → GlobalEngine → Engine → Serializable*)
Collision detection engine based on regular triangulation. Handles spheres and flat boundaries (considered as infinite-sized bounding spheres).

haveDistantTransient(*=false*)
Keep distant interactions? If True, don't delete interactions once bodies don't overlap anymore; constitutive laws will be responsible for requesting deletion. If False, delete as soon as there is no object penetration.

class yade.wrapper.**SpatialQuickSortCollider**(*inherits Collider → GlobalEngine → Engine → Serializable*)
Collider using quicksort along axes at each step, using Aabb bounds.

Its performance is lower than that of InsertionSortCollider (see Colliders' performance), but the algorithm is simple enought to make it good for checking other collider's correctness.

class yade.wrapper.**ZECollider**(*inherits Collider → GlobalEngine → Engine → Serializable*)
Collider with O(n log(n)) complexity, using a CGAL algorithm from Zomorodian and Edelsbrunner [Kettner2011] (http://www.cgal.org/Manual/beta/doc_html/cgal_manual/Box_intersection_d/Chapter_main.html)

fastestBodyMaxDist(*=-1*)
Maximum displacement of the fastest body since last run; if >= verletDist, we could get out of bboxes and will trigger full run. DEPRECATED, was only used without bins. *(auto-updated)*

numReinit(*=0*)
Cummulative number of bound array re-initialization.

periodic
Whether the collider is in periodic mode (read-only; for debugging) *(auto-updated)*

sortAxis(*=0*)
Axis for the initial contact detection.

sortThenCollide(*=false*)
Separate sorting and colliding phase; it is MUCH slower, but all interactions are processed at every step; this effectively makes the collider non-persistent, not remembering last state. (The default behavior relies on the fact that inversions during insertion sort are overlaps of bounding boxes that just started/ceased to exist, and only processes those; this makes the collider much more efficient.)

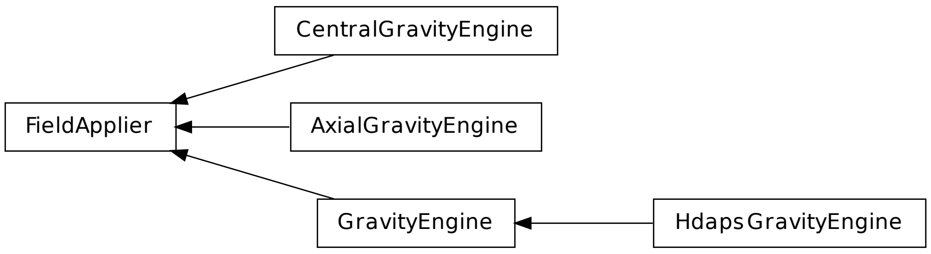
strideActive
Whether striding is active (read-only; for debugging). *(auto-updated)*

targetInterv(*=30*)
(experimental) Target number of iterations between bound update, used to define a smaller sweep distance for slower grains if >0, else always use 1*verletDist. Useful in simulations with strong velocity contrasts between slow bodies and fast bodies.

updatingDispFactor(*=-1*)
(experimental) Displacement factor used to trigger bound update: the bound is updated only if updatingDispFactor*disp>sweepDist when >0, else all bounds are updated.

verletDist(*=-.15, Automatically initialized*)
Length by which to enlarge particle bounds, to avoid running collider at every step. Stride disabled if zero. Negative value will trigger automatic computation, so that the real value will be **|verletDist|** $\times$ minimum spherical particle radius; if there are no spherical particles, it will be disabled.

7.3.4 FieldApplier



class yade.wrapper.**FieldApplier**(*inherits GlobalEngine → Engine → Serializable*)
Base for engines applying force files on particles. Not to be used directly.

class yade.wrapper.**AxialGravityEngine**(*inherits FieldApplier → GlobalEngine → Engine → Serializable*)
Apply acceleration (independent of distance) directed towards an axis.

acceleration(*=0*)
Acceleration magnitude [kgms ²]

axisDirection(*=Vector3r::UnitX()*)
direction of the gravity axis (will be normalized automatically)

axisPoint(*=Vector3r::Zero()*)
Point through which the axis is passing.

mask(*=0*)
If mask defined, only bodies with corresponding groupMask will be affected by this engine. If 0, all bodies will be affected.

class yade.wrapper.**CentralGravityEngine**(*inherits FieldApplier → GlobalEngine → Engine → Serializable*)
Engine applying acceleration to all bodies, towards a central body.

`accel`(*=0*)
: Acceleration magnitude [kgms ²]

`centralBody`(*=Body::ID_NONE*)
: The body towards which all other bodies are attracted.

`mask`(*=0*)
: If mask defined, only bodies with corresponding groupMask will be affected by this engine. If 0, all bodies will be affected.

`reciprocal`(*=false*)
: If true, acceleration will be applied on the central body as well.

class `yade.wrapper.`**`GravityEngine`**(*inherits FieldApplier → GlobalEngine → Engine → Serializable*)
Engine applying constant acceleration to all bodies. DEPRECATED, use Newton::gravity unless you need energy tracking or selective gravity application using groupMask).

`gravity`(*=Vector3r::Zero()*)
: Acceleration [kgms ²]

`mask`(*=0*)
: If mask defined, only bodies with corresponding groupMask will be affected by this engine. If 0, all bodies will be affected.

`warnOnce`(*=true*)
: For deprecation warning once.

class `yade.wrapper.`**`HdapsGravityEngine`**(*inherits GravityEngine → FieldApplier → GlobalEngine → Engine → Serializable*)
Read accelerometer in Thinkpad laptops (HDAPS and accordingly set gravity within the simulation. This code draws from hdaps-gl . See scripts/test/hdaps.py for an example.

`accel`(*=Vector2i::Zero()*)
: reading from the sysfs file

`calibrate`(*=Vector2i::Zero()*)
: Zero position; if NaN, will be read from the *hdapsDir* / calibrate.

`calibrated`(*=false*)
: Whether *calibrate* was already updated. Do not set to `True` by hand unless you also give a meaningful value for *calibrate*.

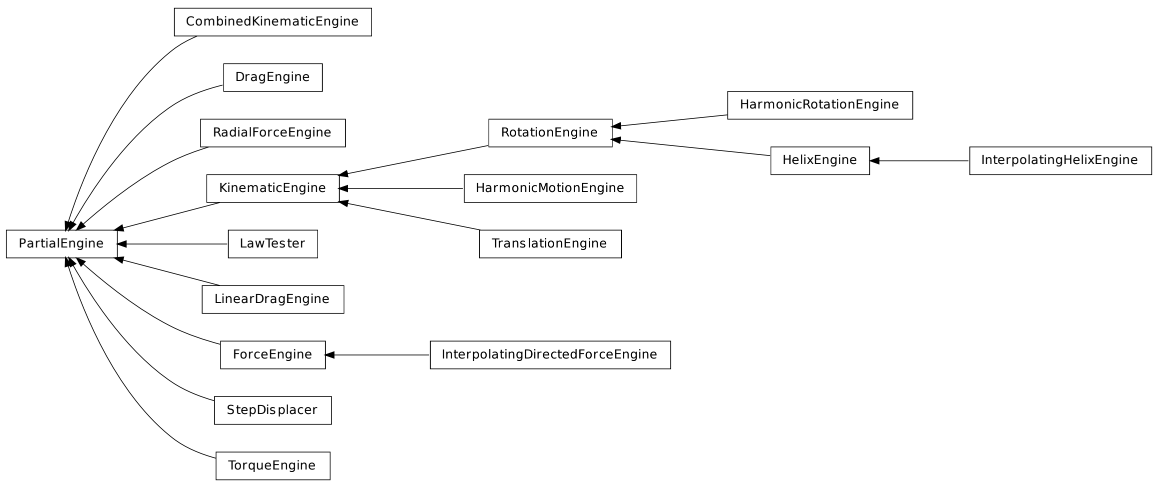
`hdapsDir`(*="/sys/devices/platform/hdaps"*)
: Hdaps directory; contains `position` (with accelerometer readings) and `calibration` (zero acceleration).

`msecUpdate`(*=50*)
: How often to update the reading.

`updateThreshold`(*=4*)
: Minimum difference of reading from the file before updating gravity, to avoid jitter.

`zeroGravity`(*=Vector3r(0, 0, -1)*)
: Gravity if the accelerometer is in flat (zero) position.

7.4 Partial engines



class `yade.wrapper.`**`PartialEngine`**(*inherits Engine → Serializable*)

Engine affecting only particular bodies in the simulation, defined by *ids*.

`ids`(*=uninitalized*)

Ids of bodies affected by this PartialEngine.

class `yade.wrapper.`**`CombinedKinematicEngine`**(*inherits PartialEngine → Engine → Serializable*)

Engine for applying combined displacements on pre-defined bodies. Constructed using `+` operator on regular KinematicEngines. The **`ids`** operated on are those of the first engine in the combination (assigned automatically).

`comb`(*=uninitalized*)

Kinematic engines that will be combined by this one, run in the order given.

class `yade.wrapper.`**`DragEngine`**(*inherits PartialEngine → Engine → Serializable*)

Apply drag force on some particles at each step, decelerating them proportionally to their linear velocities. The applied force reads

$$F_d = -\frac{v}{|v|}\frac{1}{2}\rho|v|^2 C_d A$$

where ρ is the medium density (density), v is particle's velocity, A is particle projected area (disc), C_d is the drag coefficient (0.47 for Sphere),

Note: Drag force is only applied to spherical particles, listed in ids.

`Cd`(*=0.47*)

Drag coefficient <http://en.wikipedia.org/wiki/Drag_coefficient>'_.

`Rho`(*=1.225*)

Density of the medium (fluid or air), by default - the density of the air.

class `yade.wrapper.`**`ForceEngine`**(*inherits PartialEngine → Engine → Serializable*)

Apply contact force on some particles at each step.

`force`(*=Vector3r::Zero()*)

Force to apply.

class `yade.wrapper.`**`HarmonicMotionEngine`**(*inherits KinematicEngine → PartialEngine → Engine → Serializable*)

This engine implements the harmonic oscillation of bodies. http://en.wikipedia.org/wiki/Simple_-harmonic_motion#Dynamics_of_simple_harmonic_motion

A(=*Vector3r::Zero()*)
: Amplitude [m]

f(=*Vector3r::Zero()*)
: Frequency [hertz]

fi(=*Vector3r(Mathr::PI/2.0, Mathr::PI/2.0, Mathr::PI/2.0)*)
: Initial phase [radians]. By default, the body oscillates around initial position.

class yade.wrapper.**HarmonicRotationEngine**(*inherits RotationEngine → KinematicEngine → PartialEngine → Engine → Serializable*)

This engine implements the harmonic-rotation oscillation of bodies. http://en.wikipedia.org/wiki/Simple_harmonic_motion#Dynamics_of_simple_harmonic_-motion ; please, set dynamic=False for bodies, droven by this engine, otherwise amplitude will be 2x more, than awaited.

A(=*0*)
: Amplitude [rad]

f(=*0*)
: Frequency [hertz]

fi(=*Mathr::PI/2.0*)
: Initial phase [radians]. By default, the body oscillates around initial position.

class yade.wrapper.**HelixEngine**(*inherits RotationEngine → KinematicEngine → PartialEngine → Engine → Serializable*)

Engine applying both rotation and translation, along the same axis, whence the name HelixEngine

angleTurned(=*0*)
: How much have we turned so far. *(auto-updated)* [rad]

linearVelocity(=*0*)
: Linear velocity [m/s]

class yade.wrapper.**InterpolatingDirectedForceEngine**(*inherits ForceEngine → PartialEngine → Engine → Serializable*)

Engine for applying force of varying magnitude but constant direction on subscribed bodies. times and magnitudes must have the same length, direction (normalized automatically) gives the orientation.

As usual with interpolating engines: the first magnitude is used before the first time point, last magnitude is used after the last time point. Wrap specifies whether time wraps around the last time point to the first time point.

direction(=*Vector3r::UnitX()*)
: Contact force direction (normalized automatically)

magnitudes(=*uninitalized*)
: Force magnitudes readings [N]

times(=*uninitalized*)
: Time readings [s]

wrap(=*false*)
: wrap to the beginning of the sequence if beyond the last time point

class yade.wrapper.**InterpolatingHelixEngine**(*inherits HelixEngine → RotationEngine → KinematicEngine → PartialEngine → Engine → Serializable*)

Engine applying spiral motion, finding current angular velocity by linearly interpolating in times and velocities and translation by using slope parameter.

The interpolation assumes the margin value before the first time point and last value after the last time point. If wrap is specified, time will wrap around the last times value to the first one (note that no interpolation between last and first values is done).

angularVelocities(*=uninitalized*)
List of angular velocities; manadatorily of same length as times. [rad/s]

slope(*=0*)
Axial translation per radian turn (can be negative) [m/rad]

times(*=uninitalized*)
List of time points at which velocities are given; must be increasing [s]

wrap(*=false*)
Wrap t if t>times_n, i.e. t_wrapped=t-N*(times_n-times_0)

class `yade.wrapper.`**KinematicEngine**(*inherits PartialEngine → Engine → Serializable*)
Abstract engine for applying prescribed displacement.

Note: Derived classes should override the `apply` with given list of `ids` (not `action` with PartialEngine.ids), so that they work when combined together; velocity and angular velocity of all subscribed bodies is reset before the `apply` method is called, it should therefore only increment those quantities.

class `yade.wrapper.`**LawTester**(*inherits PartialEngine → Engine → Serializable*)
Prescribe and apply deformations of an interaction in terms of local mutual displacements and rotations. The loading path is specified either using path (as sequence of 6-vectors containing generalized displacements u_x, u_y, u_z, φ_x, φ_y, φ_z) or disPath (u_x, u_y, u_z) and rotPath (φ_x, φ_y, φ_z). Time function with time values (step numbers) corresponding to points on loading path is given by pathSteps. Loading values are linearly interpolated between given loading path points, and starting zero-value (the initial configuration) is assumed for both path and pathSteps. hooks can specify python code to run when respective point on the path is reached; when the path is finished, doneHook will be run.

LawTester should be placed between InteractionLoop and NewtonIntegrator in the simulation loop, since it controls motion via setting linear/angular velocities on particles; those velocities are integrated by NewtonIntegrator to yield an actual position change, which in turn causes IGeom to be updated (and contact law applied) when InteractionLoop is executed. Constitutive law generating forces on particles will not affect prescribed particle motion, since both particles have all DoFs blocked when first used with LawTester.

LawTester uses, as much as possible, IGeom to provide useful data (such as local coordinate system), but is able to compute those independently if absent in the respective IGeom:

IGeom	#DoFs	LawTester support level
L3Geom	3	full
L6Geom	6	full
ScGeom	3	emulate local coordinate system
ScGeom6D	6	emulate local coordinate system
Dem3DofGeom	3	*not supported*

Depending on IGeom, 3 (u_x, u_y, u_z) or 6 (u_x, u_y, u_z, φ_x, φ_y, φ_z) degrees of freedom (DoFs) are controlled with LawTester, by prescribing linear and angular velocities of both particles in contact. All DoFs controlled with LawTester are orthogonal (fully decoupled) and are controlled independently.

When 3 DoFs are controlled, rotWeight controls whether local shear is applied by moving particle on arc around the other one, or by rotating without changing position; although such rotation induces mutual rotation on the interaction, it is ignored with IGeom with only 3 DoFs. When 6 DoFs are controlled, only arc-displacement is applied for shear, since otherwise mutual rotation would occur.

idWeight distributes prescribed motion between both particles (resulting local deformation is the same if `id1` is moved towards `id2` or `id2` towards `id1`). This is true only for u_x, u_y, u_z, φ_x however ; bending rotations φ_y, φ_z are nevertheless always distributed regardless of `idWeight` to both spheres in inverse proportion to their radii, so that there is no shear induced.

LawTester knows current contact deformation from 2 sources: from its own internal data (which are used for prescribing the displacement at every step), which can be accessed in uTest, and from IGeom itself (depending on which data it provides), which is stored in uGeom. These two values should be identical (disregarding numerical percision), and it is a way to test whether IGeom and related functors compute what they are supposed to compute.

LawTester-operated interactions can be rendered with GlExtra_LawTester renderer.

See scripts/test/law-test.py for an example.

`disPath`(*=uninitalized*)
: Loading path, where each Vector3 contains desired normal displacement and two components of the shear displacement (in local coordinate system, which is being tracked automatically. If shorter than rotPath, the last value is repeated.

`displIsRel`(*=true*)
: Whether displacement values in *disPath* are normalized by reference contact length (r1+r2 for 2 spheres).

`doneHook`(*=uninitalized*)
: Python command (as string) to run when end of the path is achieved. If empty, the engine will be set dead.

`hooks`(*=uninitalized*)
: Python commands to be run when the corresponding point in path is reached, before doing other things in that particular step. See also doneHook.

`idWeight`(*=1*)
: Float, usually ⟨0,1⟩, determining on how are displacements distributed between particles (0 for id1, 1 for id2); intermediate values will apply respective part to each of them. This parameter is ignored with 6-DoFs IGeom.

`pathSteps`(*=vector<int>(1, 1), (constant step)*)
: Step number for corresponding values in path; if shorter than path, distance between last 2 values is used for the rest.

`refLength`(*=0*)
: Reference contact length, for rendering only.

`renderLength`(*=0*)
: Characteristic length for the purposes of rendering, set equal to the smaller radius.

`rotPath`(*=uninitalized*)
: Rotational components of the loading path, where each item contains torsion and two bending rotations in local coordinates. If shorter than path, the last value is repeated.

`rotWeight`(*=1*)
: Float ⟨0,1⟩ determining whether shear displacement is applied as rotation or displacement on arc (0 is displacement-only, 1 is rotation-only). Not effective when mutual rotation is specified.

`step`(*=1*)
: Step number in which this engine is active; determines position in path, using pathSteps.

`trsf`(*=uninitalized*)
: Transformation matrix for the local coordinate system. *(auto-updated)*

`uGeom`(*=Vector6r::Zero()*)
: Current generalized displacements (3 displacements, 3 rotations), as stored in the interation itself. They should corredpond to uTest, otherwise a bug is indicated.

`uTest`(*=Vector6r::Zero()*)
: Current generalized displacements (3 displacements, 3 rotations), as they should be according to this LawTester. Should correspond to uGeom.

`uuPrev`(*=Vector6r::Zero()*)
: Generalized displacement values reached in the previous step, for knowing which increment to apply in the current step.

class `yade.wrapper.`**`LinearDragEngine`**(*inherits PartialEngine → Engine → Serializable*)

Apply viscous resistance or linear drag on some particles at each step, decelerating them proportionally to their linear velocities. The applied force reads

$$F_d = -b\mathbf{v}$$

where b is the linear drag, $\mathbf{v}$ is particle's velocity.

$$b = 6\pi\nu r$$

where ν is the medium viscosity, r is the Stokes radius of the particle (but in this case we accept it equal to sphere radius for simplification),

---

Note: linear drag is only applied to spherical particles, listed in ids.

`nu`(*=0.001*)
: Viscosity of the medium.

class `yade.wrapper.`**`RadialForceEngine`**(*inherits PartialEngine → Engine → Serializable*)

Apply force of given magnitude directed away from spatial axis.

`axisDir`(*=Vector3r::UnitX()*)
: Axis direction (normalized automatically)

`axisPt`(*=Vector3r::Zero()*)
: Point on axis

`fNorm`(*=0*)
: Applied force magnitude

class `yade.wrapper.`**`RotationEngine`**(*inherits KinematicEngine → PartialEngine → Engine → Serializable*)

Engine applying rotation (by setting angular velocity) to subscribed bodies. If rotateAroundZero is set, then each body is also displaced around zeroPoint.

`angularVelocity`(*=0*)
: Angular velocity. [rad/s]

`rotateAroundZero`(*=false*)
: If True, bodies will not rotate around their centroids, but rather around `zeroPoint`.

`rotationAxis`(*=Vector3r::UnitX()*)
: Axis of rotation (direction); will be normalized automatically.

`zeroPoint`(*=Vector3r::Zero()*)
: Point around which bodies will rotate if `rotateAroundZero` is True

class `yade.wrapper.`**`StepDisplacer`**(*inherits PartialEngine → Engine → Serializable*)

Apply generalized displacement (displacement or rotation) stepwise on subscribed bodies. Could be used for purposes of contact law tests (by moving one sphere compared to another), but in this case, see rather LawTester

`mov`(*=Vector3r::Zero()*)
: Linear displacement step to be applied per iteration, by addition to State.pos.

`rot`(*=Quaternionr::Identity()*)
: Rotation step to be applied per iteration (via rotation composition with State.ori).

`setVelocities`(*=false*)
: If false, positions and orientations are directly updated, without changing the speeds of concerned bodies. If true, only velocity and angularVelocity are modified. In this second case integrator is supposed to be used, so that, thanks to this Engine, the bodies will have the prescribed jump over one iteration (dt).

class `yade.wrapper.TorqueEngine`(*inherits PartialEngine → Engine → Serializable*)
Apply given torque (momentum) value at every subscribed particle, at every step.

`moment`(*=Vector3r::Zero()*)
Torque value to be applied.

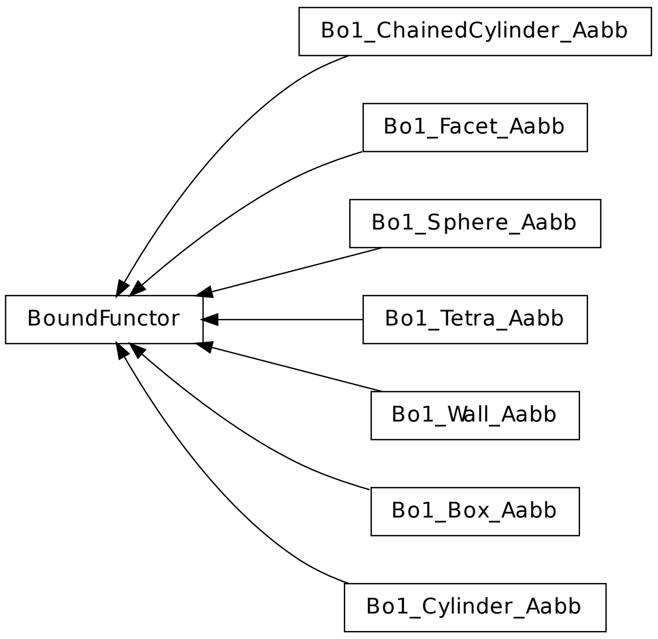
class `yade.wrapper.TranslationEngine`(*inherits KinematicEngine → PartialEngine → Engine → Serializable*)
This engine is the base class for different engines, which require any kind of motion.

`translationAxis`(*=uninitalized*)
Direction [Vector3]

`velocity`(*=uninitalized*)
Velocity [m/s]

7.5 Bounding volume creation

7.5.1 BoundFunctor



class `yade.wrapper.BoundFunctor`(*inherits Functor → Serializable*)
Functor for creating/updating Body::bound.

class `yade.wrapper.Bo1_Box_Aabb`(*inherits BoundFunctor → Functor → Serializable*)
Create/update an Aabb of a Box.

class `yade.wrapper.Bo1_ChainedCylinder_Aabb`(*inherits BoundFunctor → Functor → Serializable*)
Functor creating Aabb from ChainedCylinder.

`aabbEnlargeFactor`
Relative enlargement of the bounding box; deactivated if negative.

Note: This attribute is used to create distant interaction, but is only meaningful with an IGeomFunctor which will not simply discard such interactions: Ig2_Cylinder_Cylinder_Dem3DofGeom::distFactor / Ig2_Cylinder_Cylinder_ScGeom::interactionDetectionFactor should have the same value as aabbEnlargeFactor.

class `yade.wrapper.Bo1_Cylinder_Aabb`(*inherits BoundFunctor → Functor → Serializable*)
Functor creating Aabb from Cylinder.

`aabbEnlargeFactor`
Relative enlargement of the bounding box; deactivated if negative.

Note: This attribute is used to create distant interaction, but is only meaningful with an IGeomFunctor which will not simply discard such interactions: Ig2_Cylinder_Cylinder_-Dem3DofGeom::distFactor / Ig2_Cylinder_Cylinder_ScGeom::interactionDetectionFactor should have the same value as aabbEnlargeFactor.

class `yade.wrapper.Bo1_Facet_Aabb`(*inherits BoundFunctor → Functor → Serializable*)
Creates/updates an Aabb of a Facet.

class `yade.wrapper.Bo1_Sphere_Aabb`(*inherits BoundFunctor → Functor → Serializable*)
Functor creating Aabb from Sphere.

`aabbEnlargeFactor`
Relative enlargement of the bounding box; deactivated if negative.

Note: This attribute is used to create distant interaction, but is only meaningful with an IGeomFunctor which will not simply discard such interactions: Ig2_Sphere_Sphere_-Dem3DofGeom::distFactor / Ig2_Sphere_Sphere_ScGeom::interactionDetectionFactor should have the same value as aabbEnlargeFactor.

class `yade.wrapper.Bo1_Tetra_Aabb`(*inherits BoundFunctor → Functor → Serializable*)
Create/update Aabb of a Tetra

class `yade.wrapper.Bo1_Wall_Aabb`(*inherits BoundFunctor → Functor → Serializable*)
Creates/updates an Aabb of a Wall

7.5.2 BoundDispatcher

class `yade.wrapper.BoundDispatcher`(*inherits Dispatcher → Engine → Serializable*)
Dispatcher calling functors based on received argument type(s).

`activated`(*=true*)
Whether the engine is activated (only should be changed by the collider)

`dispFunctor`(*(Shape)arg2*) → BoundFunctor
Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

`dispMatrix`(*[(bool)names=True]*) → dict
Return dictionary with contents of the dispatch matrix.

`functors`
Functors associated with this dispatcher.

`minSweepDistFactor`(*=0.2*)
Minimal distance by which enlarge all bounding boxes; superseeds computed value of sweepDist when lower that (minSweepDistFactor x sweepDist). Updated by the collider. *(auto-updated).*

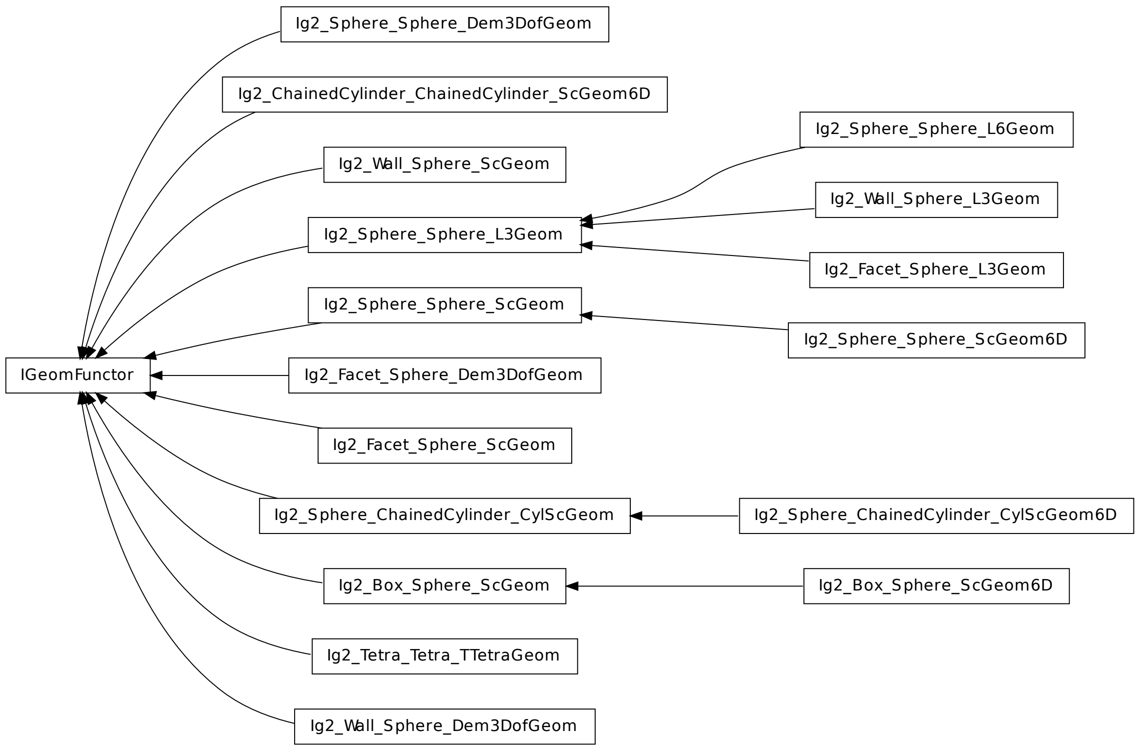
`sweepDist`(*=0*)
Distance by which enlarge all bounding boxes, to prevent collider from being run at every step (only should be changed by the collider).

`targetInterv`(*=-1*)
see InsertionSortCollider::targetInterv *(auto-updated)*

updatingDispFactor(*=-1*)
  see InsertionSortCollider::updatingDispFactor *(auto-updated)*

7.6 Interaction Geometry creation

7.6.1 IGeomFunctor


Ig2_Sphere_Sphere_Dem3DofGeom
Ig2_ChainedCylinder_ChainedCylinder_ScGeom6D
Ig2_Sphere_Sphere_L6Geom
Ig2_Wall_Sphere_ScGeom
Ig2_Wall_Sphere_L3Geom
Ig2_Sphere_Sphere_L3Geom
Ig2_Facet_Sphere_L3Geom
Ig2_Sphere_Sphere_ScGeom
Ig2_Sphere_Sphere_ScGeom6D
IGeomFunctor
Ig2_Facet_Sphere_Dem3DofGeom
Ig2_Facet_Sphere_ScGeom
Ig2_Sphere_ChainedCylinder_CylScGeom
Ig2_Sphere_ChainedCylinder_CylScGeom6D
Ig2_Box_Sphere_ScGeom
Ig2_Box_Sphere_ScGeom6D
Ig2_Tetra_Tetra_TTetraGeom
Ig2_Wall_Sphere_Dem3DofGeom

class `yade.wrapper.`**`IGeomFunctor`**(*inherits Functor → Serializable*)
  Functor for creating/updating Interaction::geom objects.

class `yade.wrapper.`**`Ig2_Box_Sphere_ScGeom`**(*inherits IGeomFunctor → Functor → Serializable*)
  Create an interaction geometry ScGeom from Box and Sphere, representing the box with a projected virtual sphere of same radius.

class `yade.wrapper.`**`Ig2_Box_Sphere_ScGeom6D`**(*inherits Ig2_Box_Sphere_ScGeom → IGeomFunctor → Functor → Serializable*)
  Create an interaction geometry ScGeom6D from Box and Sphere, representing the box with a projected virtual sphere of same radius.

class `yade.wrapper.`**`Ig2_ChainedCylinder_ChainedCylinder_ScGeom6D`**(*inherits IGeomFunctor → Functor → Serializable*)
  Create/update a ScGeom instance representing connexion between chained cylinders.

  interactionDetectionFactor(*=1*)
    Enlarge both radii by this factor (if >1), to permit creation of distant interactions.

class `yade.wrapper.`**`Ig2_Facet_Sphere_Dem3DofGeom`**(*inherits IGeomFunctor → Functor → Serializable*)
  Compute geometry of facet-sphere contact with normal and shear DOFs. As in all other Dem3DofGeom-related classes, total formulation of both shear and normal deformations is used. See Dem3DofGeom_FacetSphere for more information.

class `yade.wrapper.`**`Ig2_Facet_Sphere_L3Geom`**(*inherits Ig2_Sphere_Sphere_L3Geom → IGeomFunctor → Functor → Serializable*)

Incrementally compute L3Geom for contact between Facet and Sphere. Uses attributes of Ig2_-Sphere_Sphere_L3Geom.

class `yade.wrapper.Ig2_Facet_Sphere_ScGeom`(*inherits IGeomFunctor → Functor → Serializable*)

Create/update a ScGeom instance representing intersection of Facet and Sphere.

`shrinkFactor`(*=0, no shrinking*)

The radius of the inscribed circle of the facet is decreased by the value of the sphere's radius multipled by *shrinkFactor*. From the definition of contact point on the surface made of facets, the given surface is not continuous and becomes in effect surface covered with triangular tiles, with gap between the separate tiles equal to the sphere's radius multiplied by 2×*shrinkFactor*. If zero, no shrinking is done.

class `yade.wrapper.Ig2_Sphere_ChainedCylinder_CylScGeom`(*inherits IGeomFunctor → Functor → Serializable*)

Create/update a ScGeom instance representing intersection of two Spheres.

`interactionDetectionFactor`(*=1*)

Enlarge both radii by this factor (if >1), to permit creation of distant interactions.

class `yade.wrapper.Ig2_Sphere_ChainedCylinder_CylScGeom6D`(*inherits Ig2_Sphere_-ChainedCylinder_CylScGeom → IGeomFunctor → Functor → Serializable*)

Create/update a ScGeom6D instance representing the geometry of a contact point between two :yref:‘Spheres<Sphere>‘s, including relative rotations.

`creep`(*=false*)

Substract rotational creep from relative rotation. The rotational creep ScGeom6D::twistCreep is a quaternion and has to be updated inside a constitutive law, see for instance Law2_-ScGeom6D_CohFrictPhys_CohesionMoment.

`updateRotations`(*=false*)

Precompute relative rotations. Turning this false can speed up simulations when rotations are not needed in constitutive laws (e.g. when spheres are compressed without cohesion and moment in early stage of a triaxial test), but is not foolproof. Change this value only if you know what you are doing.

class `yade.wrapper.Ig2_Sphere_Sphere_Dem3DofGeom`(*inherits IGeomFunctor → Functor → Serializable*)

Functor handling contact of 2 spheres, producing Dem3DofGeom instance

`distFactor`(*=-1*)

Factor of sphere radius such that sphere "touch" if their centers are not further than distFactor*(r1+r2); if negative, equilibrium distance is the sum of the sphere's radii.

class `yade.wrapper.Ig2_Sphere_Sphere_L3Geom`(*inherits IGeomFunctor → Functor → Serializable*)

Functor for computing incrementally configuration of 2 Spheres stored in L3Geom; the configuration is positioned in global space by local origin $\mathbf{c}$ (contact point) and rotation matrix $\mathbf{T}$ (orthonormal transformation matrix), and its degrees of freedom are local displacement $\mathbf{u}$ (in one normal and two shear directions); with Ig2_Sphere_Sphere_L6Geom and L6Geom, there is additionally $\boldsymbol{\varphi}$. The first row of $\mathbf{T}$, i.e. local x-axis, is the contact normal noted $\mathbf{n}$ for brevity. Additionally, quasi-constant values of $\mathbf{u}_0$ (and $\boldsymbol{\varphi}_0$) are stored as shifted origins of $\mathbf{u}$ (and $\boldsymbol{\varphi}$); therefore, current value of displacement is always $\mathbf{u}^\circ - \mathbf{u}_0$.

Suppose two spheres with radii r_i, positions $\mathbf{x}_i$, velocities $\mathbf{v}_i$, angular velocities $\boldsymbol{\omega}_i$.

When there is not yet contact, it will be created if $u_N = |\mathbf{x}_2^\circ - \mathbf{x}_1^\circ| - |f_d|(r_1 + r2) < 0$, where f_d is distFactor (sometimes also called ‘‘interaction radius‘‘). If $f_d > 0$, then $\mathbf{u}_{0x}$ will be initalized to u_N, otherwise to 0. In another words, contact will be created if spheres enlarged by $|f_d|$ touch, and the ‘‘equilibrium distance" (where $\mathbf{u}_x - \mathbf{u} - 0x$ is zero) will be set to the current distance if f_d is positive, and to the geometrically-touching distance if negative.

Local axes (rows of $\mathbf{T}$) are initially defined as follows:

- local x-axis is $\boldsymbol{n} = \boldsymbol{x}_l = \widehat{\boldsymbol{x}_2 - \boldsymbol{x}_1}$;
- local y-axis positioned arbitrarily, but in a deterministic manner: aligned with the xz plane (if $\boldsymbol{n}_y < \boldsymbol{n}_z$) or xy plane (otherwise);
- local z-axis $\boldsymbol{z}_l = \boldsymbol{x}_l \times \boldsymbol{y}_l$.

If there has already been contact between the two spheres, it is updated to keep track of rigid motion of the contact (one that does not change mutual configuration of spheres) and mutual configuration changes. Rigid motion transforms local coordinate system and can be decomposed in rigid translation (affecting $\boldsymbol{c}$), and rigid rotation (affecting $\boldsymbol{T}$), which can be split in rotation $\boldsymbol{o}_r$ perpendicular to the normal and rotation $\boldsymbol{o}_t$ (“twist“) parallel with the normal:

$$\boldsymbol{o}_r^{\ominus} = \boldsymbol{n}^- \times \boldsymbol{n}^\circ.$$

Since velocities are known at previous midstep $(t - \Delta t/2)$, we consider mid-step normal

$$\boldsymbol{n}^{\ominus} = \frac{\boldsymbol{n}^- + \boldsymbol{n}^\circ}{2}.$$

For the sake of numerical stability, $\boldsymbol{n}^{\ominus}$ is re-normalized after being computed, unless prohibited by approxMask. If approxMask has the appropriate bit set, the mid-normal is not compute, and we simply use $\boldsymbol{n}^{\ominus} \approx \boldsymbol{n}^-$.

Rigid rotation parallel with the normal is

$$\boldsymbol{o}_t^{\ominus} = \boldsymbol{n}^{\ominus} \left(\boldsymbol{n}^{\ominus} \cdot \frac{\boldsymbol{\omega}_1^{\ominus} + \boldsymbol{\omega}_2^{\ominus}}{2} \right) \Delta t.$$

Branch vectors $\boldsymbol{b}_1$, $\boldsymbol{b}_2$ (connecting $\boldsymbol{x}_1^\circ$, $\boldsymbol{x}_2^\circ$ with $\boldsymbol{c}^\circ$ are computed depending on noRatch (see here).

$$\boldsymbol{b}_1 = \begin{cases} r_1 \boldsymbol{n}^\circ & \text{with } \texttt{noRatch} \\ \boldsymbol{c}^\circ - \boldsymbol{x}_1^\circ & \text{otherwise} \end{cases}$$

$$\boldsymbol{b}_2 = \begin{cases} -r_2 \boldsymbol{n}^\circ & \text{with } \texttt{noRatch} \\ \boldsymbol{c}^\circ - \boldsymbol{x}_2^\circ & \text{otherwise} \end{cases}$$

Relative velocity at $\boldsymbol{c}^\circ$ can be computed as

$$\boldsymbol{v}_r^{\ominus} = (\tilde{\boldsymbol{v}}_2^{\ominus} + \boldsymbol{\omega}_2 \times \boldsymbol{b}_2) - (\boldsymbol{v}_1 + \boldsymbol{\omega}_1 \times \boldsymbol{b}_1)$$

where $\tilde{\boldsymbol{v}}_2$ is $\boldsymbol{v}_2$ without mean-field velocity gradient in periodic boundary conditions (see Cell.homoDeform). In the numerial implementation, the normal part of incident velocity is removed (since it is computed directly) with $\boldsymbol{v}_{r2}^{\ominus} = \boldsymbol{v}_r^{\ominus} - (\boldsymbol{n}^{\ominus} \cdot \boldsymbol{v}_r^{\ominus})\boldsymbol{n}^{\ominus}$.

Any vector $\boldsymbol{a}$ expressed in global coordinates transforms during one timestep as

$$\boldsymbol{a}^\circ = \boldsymbol{a}^- + \boldsymbol{v}_r^{\ominus} \Delta t - \boldsymbol{a}^- \times \boldsymbol{o}_r^{\ominus} - \boldsymbol{a}^- \times \boldsymbol{t}_r^{\ominus}$$

where the increments have the meaning of relative shear, rigid rotation normal to $\boldsymbol{n}$ and rigid rotation parallel with $\boldsymbol{n}$. Local coordinate system orientation, rotation matrix $\boldsymbol{T}$, is updated by rows, i.e.

$$\boldsymbol{T}^\circ = \begin{pmatrix} \boldsymbol{n}_x^\circ \quad \boldsymbol{n}_y^\circ \qquad\qquad \boldsymbol{n}_z^\circ \\ \boldsymbol{T}_{1,\bullet}^- - \boldsymbol{T}_{1,\bullet}^- \times \boldsymbol{o}_r^{\ominus} - \boldsymbol{T}_{1,\bullet}^- \times \boldsymbol{o}_t^{\ominus} \\ \boldsymbol{T}_{2,\bullet}^- - \boldsymbol{T}_{2,\bullet}^- \times \boldsymbol{o}_r^{\ominus} - \boldsymbol{T}_{,\bullet}^- \times \boldsymbol{o}_t^{\ominus} \end{pmatrix}$$

This matrix is re-normalized (unless prevented by approxMask) and mid-step transformation is computed using quaternion spherical interpolation as

$$\mathbf{T}^{\ominus} = \mathrm{Slerp}\left(\mathbf{T}^{-}; \mathbf{T}^{\circ}; t = 1/2\right).$$

Depending on approxMask, this computation can be avoided by approximating $\mathbf{T}^{\ominus} = \mathbf{T}^{-}$.

Finally, current displacement is evaluated as

$$\mathbf{u}^{\circ} = u^{-} + \mathbf{T}^{\ominus}\mathbf{v}_r^{\ominus}\Delta t.$$

For the normal component, non-incremental evaluation is preferred, giving

$$\mathbf{u}_x^{\circ} = |\mathbf{x}_2^{\circ} - \mathbf{x}_1^{\circ}| - (r_1 + r_2)$$

If this functor is called for L6Geom, local rotation is updated as

$$\boldsymbol{\varphi}^{\circ} = \boldsymbol{\varphi}^{-} + \mathbf{T}^{\ominus}\Delta t(\boldsymbol{\omega}_2 - \boldsymbol{\omega}_1)$$

`approxMask`

Selectively enable geometrical approximations (bitmask); add the values for approximations to be enabled.

1	use previous transformation to transform velocities (which are known at mid-steps), instead of mid-step transformation computed as quaternion slerp at t=0.5.
2	do not take average (mid-step) normal when computing relative shear displacement, use previous value instead
4	do not re-normalize average (mid-step) normal, if used.…

By default, the mask is zero, wherefore none of these approximations is used.

`distFactor`(*=1*)

Create interaction if spheres are not futher than |distFactor|*(r1+r2). If negative, zero normal deformation will be set to be the initial value (otherwise, the geometrical distance is the 'zero' one).

`noRatch`(*=true*)

See Ig2_Sphere_Sphere_ScGeom.avoidGranularRatcheting.

`trsfRenorm`(*=100*)

How often to renormalize trsf; if non-positive, never renormalized (simulation might be unstable)

class `yade.wrapper.Ig2_Sphere_Sphere_L6Geom`(*inherits Ig2_Sphere_Sphere_L3Geom → IGeomFunctor → Functor → Serializable*)

Incrementally compute L6Geom for contact of 2 spheres.

class `yade.wrapper.Ig2_Sphere_Sphere_ScGeom`(*inherits IGeomFunctor → Functor → Serializable*)

Create/update a ScGeom instance representing the geometry of a contact point between two :yref:‘Spheres<Sphere>‘s.

`avoidGranularRatcheting`

Define relative velocity so that ratcheting is avoided. It applies for sphere-sphere contacts. It eventualy also apply for sphere-emulating interactions (i.e. convertible into the ScGeom type), if the virtual sphere's motion is defined correctly (see e.g. Ig2_Sphere_ChainedCylinder_CylScGeom.

Short explanation of what we want to avoid :

Numerical ratcheting is best understood considering a small elastic cycle at a contact between two grains : assuming b1 is fixed, impose this displacement to b2 :

1.translation dx in the normal direction

2.rotation a

3.translation -dx (back to the initial position)

4.rotation -a (back to the initial orientation)

If the branch vector used to define the relative shear in rotation×branch is not constant (typically if it is defined from the vector center→contactPoint), then the shear displacement at the end of this cycle is not zero: rotations a and -a are multiplied by branches of different lengths.

It results in a finite contact force at the end of the cycle even though the positions and orientations are unchanged, in total contradiction with the elastic nature of the problem. It could also be seen as an *inconsistent energy creation or loss*. Given that DEM simulations tend to generate oscillations around equilibrium (damped mass-spring), it can have a significant impact on the evolution of the packings, resulting for instance in slow creep in iterations under constant load.

The solution adopted here to avoid ratcheting is as proposed by McNamara and co-workers. They analyzed the ratcheting problem in detail - even though they comment on the basis of a cycle that differs from the one shown above. One will find interesting discussions in e.g. DOI 10.1103/PhysRevE.77.031304, even though solution it suggests is not fully applied here (equations of motion are not incorporating alpha, in contradiction with what is suggested by McNamara et al.).

`interactionDetectionFactor`

Enlarge both radii by this factor (if >1), to permit creation of distant interactions.

InteractionGeometry will be computed when interactionDetectionFactor*(rad1+rad2) > distance.

Note: This parameter is functionally coupled with Bo1_Sphere_Aabb::aabbEnlargeFactor, which will create larger bounding boxes and should be of the same value.

class `yade.wrapper.Ig2_Sphere_Sphere_ScGeom6D`(*inherits Ig2_Sphere_Sphere_ScGeom → IGeomFunctor → Functor → Serializable*)

Create/update a ScGeom6D instance representing the geometry of a contact point between two :yref:‘Spheres<Sphere>‘s, including relative rotations.

`creep`(*=false*)

Substract rotational creep from relative rotation. The rotational creep ScGeom6D::twistCreep is a quaternion and has to be updated inside a constitutive law, see for instance Law2_ScGeom6D_CohFrictPhys_CohesionMoment.

`updateRotations`(*=true*)

Precompute relative rotations. Turning this false can speed up simulations when rotations are not needed in constitutive laws (e.g. when spheres are compressed without cohesion and moment in early stage of a triaxial test), but is not foolproof. Change this value only if you know what you are doing.

class `yade.wrapper.Ig2_Tetra_Tetra_TTetraGeom`(*inherits IGeomFunctor → Functor → Serializable*)

Create/update geometry of collision between 2 tetrahedra (TTetraGeom instance)

class `yade.wrapper.Ig2_Wall_Sphere_Dem3DofGeom`(*inherits IGeomFunctor → Functor → Serializable*)

Create/update contact of Wall and Sphere (Dem3DofGeom_WallSphere instance)

class `yade.wrapper.Ig2_Wall_Sphere_L3Geom`(*inherits Ig2_Sphere_Sphere_L3Geom → IGeomFunctor → Functor → Serializable*)

Incrementally compute L3Geom for contact between Wall and Sphere. Uses attributes of Ig2_Sphere_Sphere_L3Geom.

class `yade.wrapper.Ig2_Wall_Sphere_ScGeom`(*inherits IGeomFunctor → Functor → Serializable*)

Create/update a ScGeom instance representing intersection of Wall and Sphere.

`noRatch`(*=true*)

Avoid granular ratcheting

7.6.2 IGeomDispatcher

class `yade.wrapper.IGeomDispatcher`(*inherits Dispatcher → Engine → Serializable*)

Dispatcher calling functors based on received argument type(s).

`dispFunctor`(*(Shape)arg2*, *(Shape)arg3*) → IGeomFunctor

Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

`dispMatrix`([*(bool)names=True*]) → dict

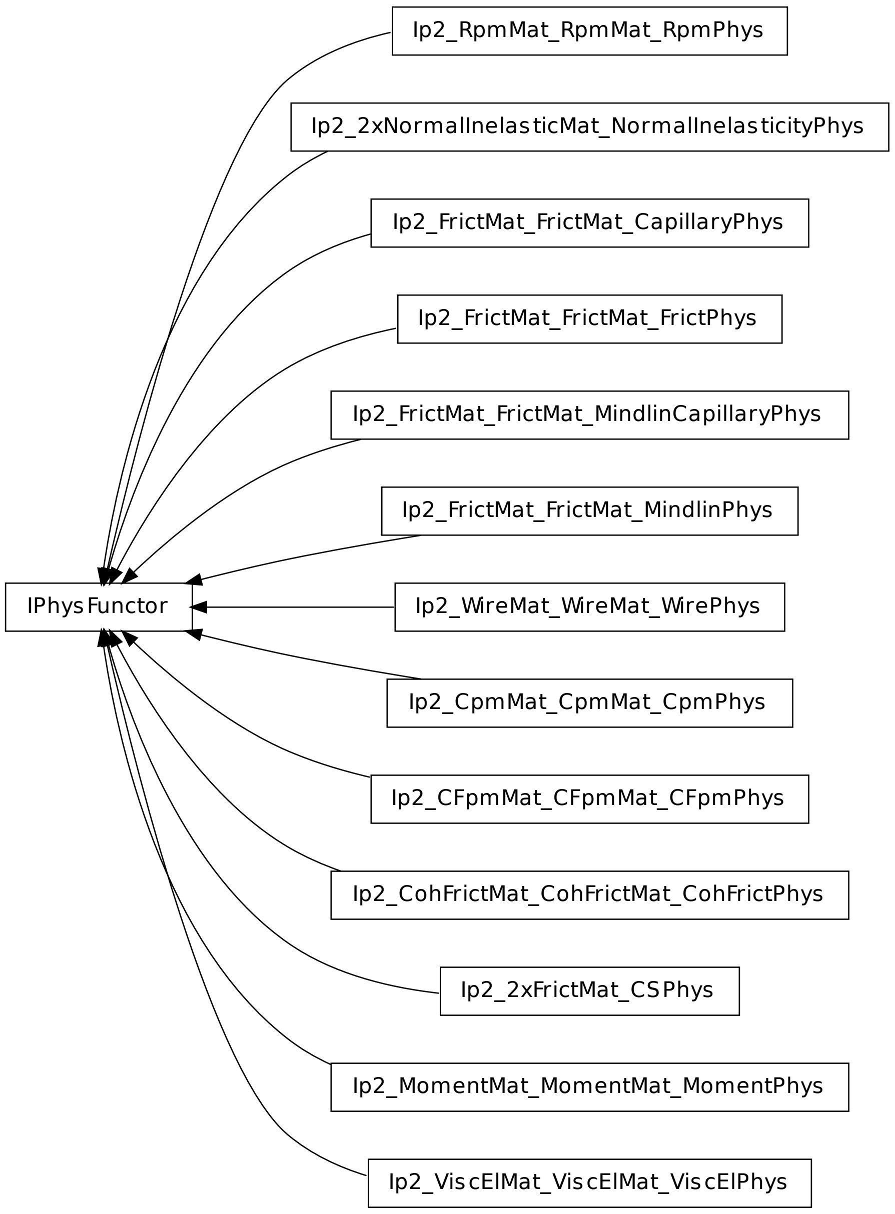
Return dictionary with contents of the dispatch matrix.

`functors`

Functors associated with this dispatcher.

7.7 Interaction Physics creation

7.7.1 IPhysFunctor


Ip2_RpmMat_RpmMat_RpmPhys
Ip2_2xNormalInelasticMat_NormalInelasticityPhys
Ip2_FrictMat_FrictMat_CapillaryPhys
Ip2_FrictMat_FrictMat_FrictPhys
Ip2_FrictMat_FrictMat_MindlinCapillaryPhys
Ip2_FrictMat_FrictMat_MindlinPhys
IPhysFunctor
Ip2_WireMat_WireMat_WirePhys
Ip2_CpmMat_CpmMat_CpmPhys
Ip2_CFpmMat_CFpmMat_CFpmPhys
Ip2_CohFrictMat_CohFrictMat_CohFrictPhys
Ip2_2xFrictMat_CSPhys
Ip2_MomentMat_MomentMat_MomentPhys
Ip2_ViscElMat_ViscElMat_ViscElPhys

class `yade.wrapper.IPhysFunctor`(*inherits Functor → Serializable*)
Functor for creating/updating Interaction::phys objects.

class `yade.wrapper.Ip2_2xFrictMat_CSPhys`(*inherits IPhysFunctor → Functor → Serializable*)
Functor creating CSPhys from two FrictMat. See Law2_Dem3Dof_CSPhys_CundallStrack for details.

class `yade.wrapper.Ip2_2xNormalInelasticMat_NormalInelasticityPhys`(*inherits IPhysFunctor → Functor → Serializable*)
The RelationShips for using Law2_ScGeom6D_NormalInelasticityPhys_NormalInelasticity

In these RelationShips all the attributes of the interactions (which are of NormalInelasticityPhys type) are computed.

> **Warning:** as in the others Ip2 functors, most of the attributes are computed only once, when the interaction is new.

betaR(*=0.12*)
Parameter for computing the torque-stifness : T-stifness = betaR * Rmoy^2

class `yade.wrapper.Ip2_CFpmMat_CFpmMat_CFpmPhys`(*inherits IPhysFunctor → Functor → Serializable*)
Converts 2 CFpmmat instances to CFpmPhys with corresponding parameters.

Alpha(*=0*)
Defines the ratio ks/kn.

Beta(*=0*)
Defines the ratio kr/(ks*meanRadius^2) to compute the resistive moment in rotation. [-]

cohesion(*=0*)
Defines the maximum admissible tangential force in shear FsMax=cohesion*crossSection. [Pa]

cohesiveTresholdIteration(*=1*)
Should new contacts be cohesive? They will before this iter, they won't afterward.

eta(*=0*)
Defines the maximum admissible resistive moment in rotation MtMax=eta*meanRadius*Fn. [-]

strengthSoftening(*=0*)
Defines the softening when Dtensile is reached to avoid explosion of the contact. Typically, when D > Dtensile, Fn=FnMax - (kn/strengthSoftening)*(Dtensile-D). [-]

tensileStrength(*=0*)
Defines the maximum admissible normal force in traction FnMax=tensileStrength*crossSection. [Pa]

useAlphaBeta(*=false*)
If true, stiffnesses are computed based on Alpha and Beta.

class `yade.wrapper.Ip2_CohFrictMat_CohFrictMat_CohFrictPhys`(*inherits IPhysFunctor → Functor → Serializable*)
Generates cohesive-frictional interactions with moments. Used in the contact law Law2_ScGeom6D_CohFrictPhys_CohesionMoment.

setCohesionNow(*=false*)
If true, assign cohesion to all existing contacts in current time-step. The flag is turned false automatically, so that assignment is done in the current timestep only.

setCohesionOnNewContacts(*=false*)
If true, assign cohesion at all new contacts. If false, only existing contacts can be cohesive (also see Ip2_CohFrictMat_CohFrictMat_CohFrictPhys::setCohesionNow), and new contacts are only frictional.

class `yade.wrapper.Ip2_CpmMat_CpmMat_CpmPhys`(*inherits IPhysFunctor → Functor → Serializable*)
Convert 2 CpmMat instances to CpmPhys with corresponding parameters. Uses simple (arithmetic) averages if material are different. Simple copy of parameters is performed if the material is shared between both particles. See *cpm-model* for detals.

cohesiveThresholdIter(*=10*)
Should new contacts be cohesive? They will before this iter#, they will not be afterwards. If 0, they will never be. If negative, they will always be created as cohesive (10 by default).

class `yade.wrapper.Ip2_FrictMat_FrictMat_CapillaryPhys`(*inherits IPhysFunctor → Functor → Serializable*)
RelationShips to use with Law2_ScGeom_CapillaryPhys_Capillarity

In these RelationShips all the interaction attributes are computed.

Warning: as in the others Ip2 functors, most of the attributes are computed only once, when the interaction is new.

class `yade.wrapper.Ip2_FrictMat_FrictMat_FrictPhys`(*inherits IPhysFunctor → Functor → Serializable*)

Create a FrictPhys from two FrictMats. The compliance of one sphere under symetric point loads is defined here as 1/(E.r), with E the stiffness of the sphere and r its radius, and corresponds to a compliance 1/(2.E.r)=1/(E.D) from each contact point. The compliance of the contact itself will be the sum of compliances from each sphere, i.e. 1/(E.D1)+1/(E.D2) in the general case, or 1/(E.r) in the special case of equal sizes. Note that summing compliances corresponds to an harmonic average of stiffnesss, which is how kn is actually computed in the Ip2_FrictMat_-FrictMat_FrictPhys functor.

The shear stiffness ks of one sphere is defined via the material parameter ElastMat::poisson, as ks=poisson*kn, and the resulting shear stiffness of the interaction will be also an harmonic average.

frictAngle(*=uninitalized*)
: Instance of MatchMaker determining how to compute interaction's friction angle. If `None`, minimum value is used.

class `yade.wrapper.Ip2_FrictMat_FrictMat_MindlinCapillaryPhys`(*inherits IPhysFunctor → Functor → Serializable*)

RelationShips to use with Law2_ScGeom_CapillaryPhys_Capillarity

In these RelationShips all the interaction attributes are computed.

Warning: as in the others Ip2 functors, most of the attributes are computed only once, when the interaction is new.

betan(*=uninitalized*)
: Normal viscous damping coefficient β_n.

betas(*=uninitalized*)
: Shear viscous damping coefficient β_s.

en(*=uninitalized*)
: Normal coefficient of restitution e_n.

es(*=uninitalized*)
: Shear coefficient of restitution e_s.

eta(*=0.0*)
: Coefficient to determine the plastic bending moment

gamma(*=0.0*)
: Surface energy parameter [J/m^2] per each unit contact surface, to derive DMT formulation from HM

krot(*=0.0*)
: Rotational stiffness for moment contact law

ktwist(*=0.0*)
: Torsional stiffness for moment contact law

class `yade.wrapper.Ip2_FrictMat_FrictMat_MindlinPhys`(*inherits IPhysFunctor → Functor → Serializable*)

Calculate some physical parameters needed to obtain the normal and shear stiffnesses according to the Hertz-Mindlin's formulation (as implemented in PFC).

Viscous parameters can be specified either using coefficients of restitution (e_n, e_s) or viscous damping coefficient (β_n, β_s). The following rules apply: #. If the β_n (β_s) coefficient is given, it is assigned to MindlinPhys.betan (MindlinPhys.betas) directly. #. If e_n is given, MindlinPhys.betan is computed using $\beta_n = -(\log e_n)/\sqrt{\pi^2 + (\log e_n)^2}$. The same applies to e_s, MindlinPhys.betas. #. It is an error (exception) to specify both e_n and β_n (e_s and β_s). #. If neither e_n nor β_n is given, zero value for MindlinPhys.betan is used; there will be no viscous effects. #.If neither e_s nor β_s is given, the value of MindlinPhys.betan is used for MindlinPhys.betas as well.

The e_n, β_n, e_s, β_s are MatchMaker objects; they can be constructed from float values to always return constant value.

See scripts/test/shots.py for an example of specifying e_n based on combination of parameters.

betan(*=uninitalized*)
Normal viscous damping coefficient β_n.

betas(*=uninitalized*)
Shear viscous damping coefficient β_s.

en(*=uninitalized*)
Normal coefficient of restitution e_n.

es(*=uninitalized*)
Shear coefficient of restitution e_s.

eta(*=0.0*)
Coefficient to determine the plastic bending moment

gamma(*=0.0*)
Surface energy parameter [J/m^2] per each unit contact surface, to derive DMT formulation from HM

krot(*=0.0*)
Rotational stiffness for moment contact law

ktwist(*=0.0*)
Torsional stiffness for moment contact law

class `yade.wrapper.`**`Ip2_MomentMat_MomentMat_MomentPhys`**(*inherits IPhysFunctor → Functor → Serializable*)
Create MomentPhys from 2 instances of MomentMat.

1. If boolean userInputStiffness=true & useAlphaBeta=false, users can input Knormal, Kshear and Krotate directly. Then, kn,ks and kr will be equal to these values, rather than calculated E and v.
2. If boolean userInputStiffness=true & useAlphaBeta=true, users input Knormal, Alpha and Beta. Then ks and kr are calculated from alpha & beta respectively.
3. If both are false, it calculates kn and ks are calculated from E and v, whilst kr = 0.

Alpha(*=0*)
Ratio of Ks/Kn

Beta(*=0*)
Ratio to calculate Kr

Knormal(*=0*)
Allows user to input stiffness properties from triaxial test. These will be passed to MomentPhys or NormShearPhys

Krotate(*=0*)
Allows user to input stiffness properties from triaxial test. These will be passed to MomentPhys or NormShearPhys

Kshear(*=0*)
Allows user to input stiffness properties from triaxial test. These will be passed to MomentPhys or NormShearPhys

useAlphaBeta(*=false*)
for users to choose whether to input stiffness directly or use ratios to calculate Ks/Kn

userInputStiffness(*=false*)
for users to choose whether to input stiffness directly or use ratios to calculate Ks/Kn

class `yade.wrapper.`**`Ip2_RpmMat_RpmMat_RpmPhys`**(*inherits IPhysFunctor → Functor → Serializable*)

Convert 2 RpmMat instances to RpmPhys with corresponding parameters.

`initDistance`(*=0*)

Initial distance between spheres at the first step.

class `yade.wrapper.`**`Ip2_ViscElMat_ViscElMat_ViscElPhys`**(*inherits IPhysFunctor → Functor → Serializable*)

Convert 2 instances of ViscElMat to ViscElPhys using the rule of consecutive connection.

class `yade.wrapper.`**`Ip2_WireMat_WireMat_WirePhys`**(*inherits IPhysFunctor → Functor → Serializable*)

Converts 2 WireMat instances to WirePhys with corresponding parameters.

`linkThresholdIteration`(*=1*)

Iteration to create the link.

7.7.2 IPhysDispatcher

class `yade.wrapper.`**`IPhysDispatcher`**(*inherits Dispatcher → Engine → Serializable*)

Dispatcher calling functors based on received argument type(s).

`dispFunctor`(*(Material)arg2*, *(Material)arg3*) → IPhysFunctor

Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

`dispMatrix`([*(bool)names=True*]) → dict

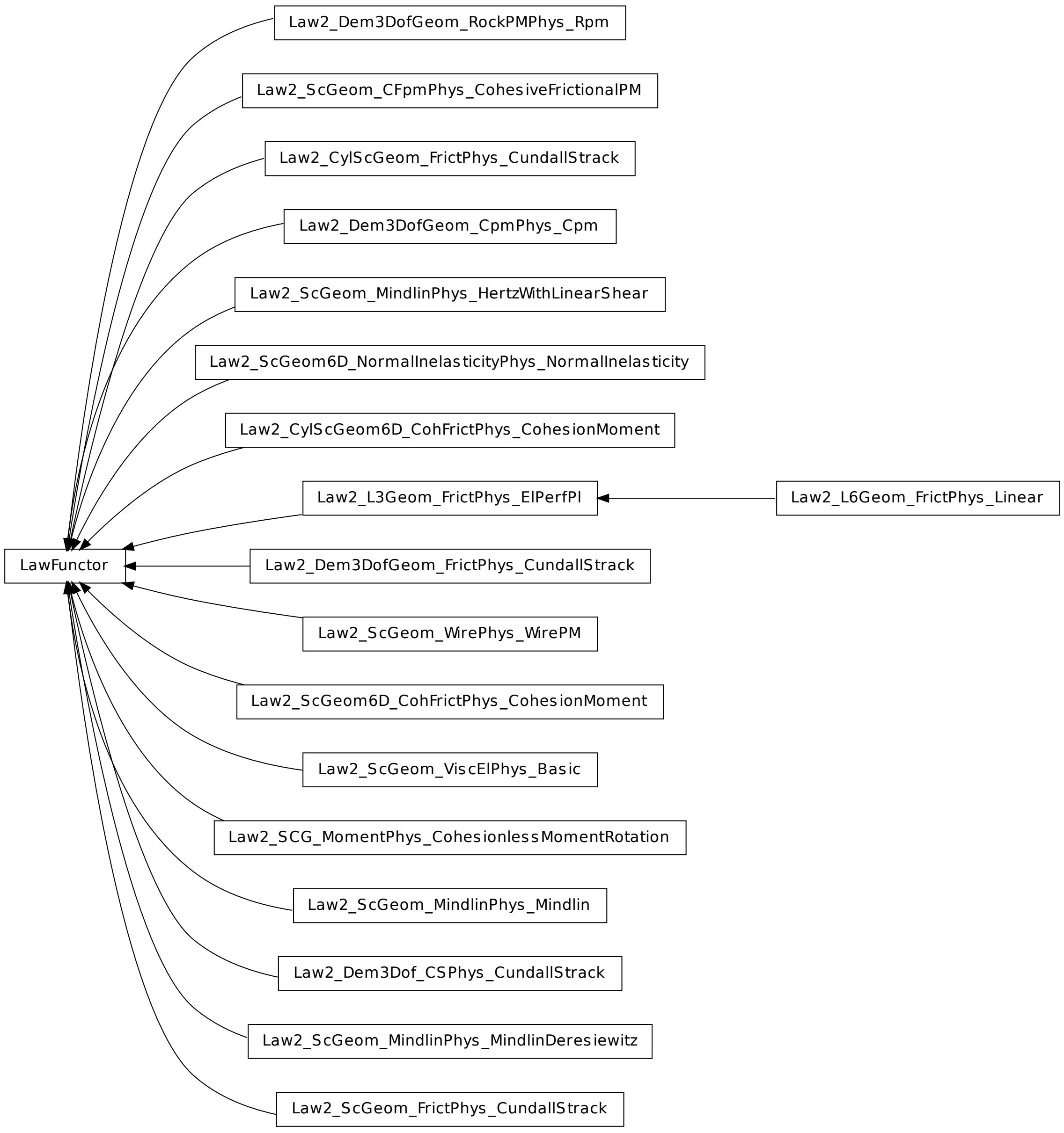
Return dictionary with contents of the dispatch matrix.

`functors`

Functors associated with this dispatcher.

7.8 Constitutive laws

7.8.1 LawFunctor


Law2_Dem3DofGeom_RockPMPhys_Rpm
Law2_ScGeom_CFpmPhys_CohesiveFrictionalPM
Law2_CylScGeom_FrictPhys_CundallStrack
Law2_Dem3DofGeom_CpmPhys_Cpm
Law2_ScGeom_MindlinPhys_HertzWithLinearShear
Law2_ScGeom6D_NormalInelasticityPhys_NormalInelasticity
Law2_CylScGeom6D_CohFrictPhys_CohesionMoment
Law2_L3Geom_FrictPhys_ElPerfPl
Law2_L6Geom_FrictPhys_Linear
LawFunctor
Law2_Dem3DofGeom_FrictPhys_CundallStrack
Law2_ScGeom_WirePhys_WirePM
Law2_ScGeom6D_CohFrictPhys_CohesionMoment
Law2_ScGeom_ViscElPhys_Basic
Law2_SCG_MomentPhys_CohesionlessMomentRotation
Law2_ScGeom_MindlinPhys_Mindlin
Law2_Dem3Dof_CSPhys_CundallStrack
Law2_ScGeom_MindlinPhys_MindlinDeresiewitz
Law2_ScGeom_FrictPhys_CundallStrack

class `yade.wrapper.LawFunctor`(*inherits Functor → Serializable*)

Functor for applying constitutive laws on interactions.

class `yade.wrapper.Law2_CylScGeom6D_CohFrictPhys_CohesionMoment`(*inherits LawFunctor → Functor → Serializable*)

Law for linear compression, and Mohr-Coulomb plasticity surface without cohesion. This law implements the classical linear elastic-plastic law from [CundallStrack1979] (see also [Pfc3dManual30]). The normal force is (with the convention of positive tensile forces) $F_n = \min(k_n u_n, 0)$. The shear force is $F_s = k_s u_s$, the plasticity condition defines the maximum value of the shear force : $F_s^{\max} = F_n \tan(\varphi)$, with φ the friction angle.

Note: This law uses ScGeom; there is also functionally equivalent Law2_Dem3DofGeom_-FrictPhys_CundallStrack, which uses Dem3DofGeom (sphere-box interactions are not implemented for the latest).

Note: This law is well tested in the context of triaxial simulation, and has been used for a number of published results (see e.g. [Scholtes2009b] and other papers from the same authors). It is generalised by Law2_ScGeom6D_CohFrictPhys_CohesionMoment, which adds cohesion and moments at contact.

---

always_use_moment_law(*=false*)
If true, use bending/twisting moments at all contacts. If false, compute moments only for cohesive contacts.

creep_viscosity(*=1*)
creep viscosity [Pa.s/m]. probably should be moved to Ip2_CohFrictMat_CohFrictMat_-CohFrictPhys...

neverErase(*=false*)
Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. Law2_ScGeom_CapillaryPhys_Capillarity)

shear_creep(*=false*)
activate creep on the shear force, using CohesiveFrictionalContactLaw::creep_viscosity.

twist_creep(*=false*)
activate creep on the twisting moment, using CohesiveFrictionalContactLaw::creep_viscosity.

useIncrementalForm(*=false*)
use the incremental formulation to compute bending and twisting moments. Creep on the twisting moment is not included in such a case.

class `yade.wrapper.`**`Law2_CylScGeom_FrictPhys_CundallStrack`**(*inherits LawFunctor → Functor → Serializable*)
Law for linear compression, and Mohr-Coulomb plasticity surface without cohesion. This law implements the classical linear elastic-plastic law from [CundallStrack1979] (see also [Pfc3dManual30]). The normal force is (with the convention of positive tensile forces) $F_n = \min(k_n u_n, 0)$. The shear force is $F_s = k_s u_s$, the plasticity condition defines the maximum value of the shear force : $F_s^{\max} = F_n \tan(\varphi)$, with φ the friction angle.

Note: This law uses ScGeom; there is also functionally equivalent Law2_Dem3DofGeom_-FrictPhys_CundallStrack, which uses Dem3DofGeom (sphere-box interactions are not implemented for the latest).

Note: This law is well tested in the context of triaxial simulation, and has been used for a number of published results (see e.g. [Scholtes2009b] and other papers from the same authors). It is generalised by Law2_ScGeom6D_CohFrictPhys_CohesionMoment, which adds cohesion and moments at contact.

---

neverErase(*=false*)
Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. Law2_ScGeom_CapillaryPhys_Capillarity)

class `yade.wrapper.`**`Law2_Dem3DofGeom_CpmPhys_Cpm`**(*inherits LawFunctor → Functor → Serializable*)
Constitutive law for the *cpm-model*.

epsSoft(*=-3e-3, approximates confinement -20MPa precisely, -100MPa a little over, -200 and -400 are OK (secant)*)
Strain at which softening in compression starts (non-negative to deactivate)

funcG(*(float)epsCrackOnset, (float)epsFracture*[*, (bool)neverDamage=False*[*, (int)damLaw=1*]]) → float
Damage evolution law, evaluating the ω parameter. κ_D is historically maximum strain, *ep-*

sCrackOnset (ε_0) = CpmPhys.epsCrackOnset, *epsFracture* = CpmPhys.epsFracture; if *neverDamage* is `True`, the value returned will always be 0 (no damage). TODO

`omegaThreshold`(*=1., >=1. to deactivate, i.e. never delete any contacts*)
: damage after which the contact disappears (<1), since omega reaches 1 only for strain →+∞

`relKnSoft`(*=.3*)
: Relative rigidity of the softening branch in compression (0=perfect elastic-plastic, <0 softening, >0 hardening)

`yieldEllipseShift`(*=NaN*)
: horizontal scaling of the ellipse (shifts on the +x axis as interactions with +y are given)

`yieldLogSpeed`(*=.1*)
: scaling in the logarithmic yield surface (should be <1 for realistic results; >=0 for meaningful results)

`yieldSigmaTMagnitude`(*(float)sigmaN, (float)omega, (float)undamagedCohesion, (float)tanFrictionAngle*) → float
: Return radius of yield surface for given material and state parameters; uses attributes of the current instance (*yieldSurfType* etc), change them before calling if you need that.

`yieldSurfType`(*=2*)
: yield function: 0: mohr-coulomb (original); 1: parabolic; 2: logarithmic, 3: log+lin_tension, 4: elliptic, 5: elliptic+log

class `yade.wrapper.`**`Law2_Dem3DofGeom_FrictPhys_CundallStrack`**(*inherits LawFunctor → Functor → Serializable*)
: Constitutive law for linear compression, no tension, and linear plasticity surface.

  No longer maintained and linking to known bugs; :consider using yref:*Law2_ScGeom_FrictPhys_CundallStrack*.

class `yade.wrapper.`**`Law2_Dem3DofGeom_RockPMPhys_Rpm`**(*inherits LawFunctor → Functor → Serializable*)
: Constitutive law for the Rpm model

class `yade.wrapper.`**`Law2_Dem3Dof_CSPhys_CundallStrack`**(*inherits LawFunctor → Functor → Serializable*)
: Basic constitutive law published originally by Cundall&Strack; it has normal and shear stiffnesses (Kn, Kn) and dry Coulomb friction. Operates on associated Dem3DofGeom and CSPhys instances.

class `yade.wrapper.`**`Law2_L3Geom_FrictPhys_ElPerfPl`**(*inherits LawFunctor → Functor → Serializable*)
: Basic law for testing L3Geom; it bears no cohesion (unless *noBreak* is `True`), and plastic slip obeys the Mohr-Coulomb criterion (unless *noSlip* is `True`).

  `noBreak`(*=false*)
  : Do not break contacts when particles separate.

  `noSlip`(*=false*)
  : No plastic slipping.

class `yade.wrapper.`**`Law2_L6Geom_FrictPhys_Linear`**(*inherits Law2_L3Geom_FrictPhys_ElPerfPl → LawFunctor → Functor → Serializable*)
: Basic law for testing L6Geom – linear in both normal and shear sense, without slip or breakage.

  `charLen`(*=1*)
  : Characteristic length with the meaning of the stiffness ratios bending/shear and torsion/normal.

class `yade.wrapper.`**`Law2_SCG_MomentPhys_CohesionlessMomentRotation`**(*inherits LawFunctor → Functor → Serializable*)
: Contact law based on Plassiard et al. (2009) : A spherical discrete element model: calibration procedure and incremental response. The functionality has been verified with results in the paper.

  The contribution of stiffnesses are scaled according to the radius of the particle, as implemented in that paper.

See also associated classes MomentMat, Ip2_MomentMat_MomentMat_MomentPhys, MomentPhys.

Note: This constitutive law can be used with triaxial test, but the following significant changes in code have to be made: Ip2_MomentMat_MomentMat_MomentPhys and Law2_SCG_MomentPhys_CohesionlessMomentRotation have to be added. Since it uses ScGeom, it uses boxes rather than facets. Spheres and boxes have to be changed to MomentMat rather than FrictMat.

---

preventGranularRatcheting(*=false*)
??

class **yade.wrapper.Law2_ScGeom6D_CohFrictPhys_CohesionMoment**(*inherits LawFunctor → Functor → Serializable*)

Law for linear traction-compression-bending-twisting, with cohesion+friction and Mohr-Coulomb plasticity surface. This law adds adhesion and moments to Law2_ScGeom_FrictPhys_CundallStrack.

The normal force is (with the convention of positive tensile forces) $F_n = min(k_n * u_n, a_n)$, with a_n the normal adhesion. The shear force is $F_s = k_s * u_s$, the plasticity condition defines the maximum value of the shear force, by default $F_s^{max} = F_n * tan(\varphi) + a_s$, with φ the friction angle and a_n the shear adhesion. If CohFrictPhys::cohesionDisableFriction is True, friction is ignored as long as adhesion is active, and the maximum shear force is only $F_s^{max} = a_s$.

If the maximum tensile or maximum shear force is reached and CohFrictPhys::fragile =True (default), the cohesive link is broken, and a_n, a_s are set back to zero. If a tensile force is present, the contact is lost, else the shear strength is $F_s^{max} = F_n * tan(\varphi)$. If CohFrictPhys::fragile =False (in course of implementation), the behaviour is perfectly plastic, and the shear strength is kept constant.

If Law2_ScGeom6D_CohFrictPhys_CohesionMoment::momentRotationLaw =True, bending and twisting moments are computed using a linear law with moduli respectively k_t and k_r (the two values are the same currently), so that the moments are : $M_b = k_b * \Theta_b$ and $M_t = k_t * \Theta_t$, with $\Theta_{b,t}$ the relative rotations between interacting bodies. There is no maximum value of moments in the current implementation, though they could be added in the future.

Creep at contact is implemented in this law, as defined in [Hassan2010]. If activated, there is a viscous behaviour of the shear and twisting components, and the evolution of the elastic parts of shear displacement and relative twist is given by $du_{s,e}/dt = -F_s/\nu_s$ and $d\Theta_{t,e}/dt = -M_t/\nu_t$.

Note: Periodicity is not handled yet in this law.

---

always_use_moment_law(*=false*)
If true, use bending/twisting moments at all contacts. If false, compute moments only for cohesive contacts.

creep_viscosity(*=1*)
creep viscosity [Pa.s/m]. probably should be moved to Ip2_CohFrictMat_CohFrictMat_CohFrictPhys...

neverErase(*=false*)
Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. Law2_ScGeom_CapillaryPhys_Capillarity)

normElastEnergy() → float
Compute normal elastic energy.

shearElastEnergy() → float
Compute shear elastic energy.

shear_creep(*=false*)
activate creep on the shear force, using CohesiveFrictionalContactLaw::creep_viscosity.

twist_creep(*=false*)
activate creep on the twisting moment, using CohesiveFrictionalContactLaw::creep_viscosity.

useIncrementalForm(*=false*)
: use the incremental formulation to compute bending and twisting moments. Creep on the twisting moment is not included in such a case.

class yade.wrapper.**Law2_ScGeom6D_NormalInelasticityPhys_NormalInelasticity**(*inherits LawFunctor → Functor → Serializable*)
: Contact law used to simulate granular filler in rock joints [Duriez2009a], [Duriez2011]. It includes possibility of cohesion, moment transfer and inelastic compression behaviour (to reproduce the normal inelasticity observed for rock joints, for the latter).

The moment transfer relation corresponds to the adaptation of the work of Plassiard & Belheine (see in [DeghmReport2006] for example), which was realized by J. Kozicki, and is now coded in ScGeom6D.

As others LawFunctor, it uses pre-computed data of the interactions (rigidities, friction angles -with their tan()-, orientations of the interactions); this work is done here in Ip2_2xNormalInelasticMat_NormalInelasticityPhys.

To use this you should also use NormalInelasticMat as material type of the bodies.

The effects of this law are illustrated in examples/normalInelasticityTest.py

momentAlwaysElastic(*=false*)
: boolean, true=> the part of the contact torque (caused by relative rotations, which is computed only if momentRotationLaw..) is not limited by a plastic threshold

momentRotationLaw(*=true*)
: boolean, true=> computation of a torque (against relative rotation) exchanged between particles

class yade.wrapper.**Law2_ScGeom_CFpmPhys_CohesiveFrictionalPM**(*inherits LawFunctor → Functor → Serializable*)
: Constitutive law for the CFpm model.

preventGranularRatcheting(*=true*)
: If true rotations are computed such as granular ratcheting is prevented. See article [Alonso2004], pg. 3-10 – and a lot more papers from the same authors).

class yade.wrapper.**Law2_ScGeom_FrictPhys_CundallStrack**(*inherits LawFunctor → Functor → Serializable*)
: Law for linear compression, and Mohr-Coulomb plasticity surface without cohesion. This law implements the classical linear elastic-plastic law from [CundallStrack1979] (see also [Pfc3dManual30]). The normal force is (with the convention of positive tensile forces) $F_n = \min(k_n u_n, 0)$. The shear force is $F_s = k_s u_s$, the plasticity condition defines the maximum value of the shear force : $F_s^{max} = F_n \tan(\varphi)$, with φ the friction angle.

This law is well tested in the context of triaxial simulation, and has been used for a number of published results (see e.g. [Scholtes2009b] and other papers from the same authors). It is generalised by Law2_ScGeom6D_CohFrictPhys_CohesionMoment, which adds cohesion and moments at contact.

elasticEnergy() → float
: Compute and return the total elastic energy in all "FrictPhys" contacts

initPlasticDissipation(*(float)arg2*) → None
: Initialize cummulated plastic dissipation to a value (0 by default).

neverErase(*=false*)
: Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. Law2_ScGeom_CapillaryPhys_Capillarity)

plasticDissipation() → float
: Total energy dissipated in plastic slips at all FrictPhys contacts. Computed only if Law2_-ScGeom_FrictPhys_CundallStrack::traceEnergy is true.

sphericalBodies(*=true*)
: If true, compute branch vectors from radii (faster), else use contactPoint-position. Turning this flag true is safe for sphere-sphere contacts and a few other specific cases. It will give wrong values of torques on facets or boxes.

traceEnergy(*=false*)
: Define the total energy dissipated in plastic slips at all contacts. This will trace only plastic energy in this law, see O.trackEnergy for a more complete energies tracing

class yade.wrapper.**Law2_ScGeom_MindlinPhys_HertzWithLinearShear**(*inherits LawFunctor → Functor → Serializable*)

Constitutive law for the Hertz formulation (using MindlinPhys.kno) and linear beahvior in shear (using MindlinPhys.kso for stiffness and FrictPhys.tangensOfFrictionAngle).

Note: No viscosity or damping. If you need those, look at Law2_ScGeom_MindlinPhys_Mindlin, which also includes non-linear Mindlin shear.

neverErase(*=false*)
: Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. Law2_ScGeom_CapillaryPhys_Capillarity)

nonLin(*=0*)
: Shear force nonlinearity (the value determines how many features of the non-linearity are taken in account). 1: ks as in HM 2: shearElastic increment computed as in HM 3. granular ratcheting disabled.

class yade.wrapper.**Law2_ScGeom_MindlinPhys_Mindlin**(*inherits LawFunctor → Functor → Serializable*)

Constitutive law for the Hertz-Mindlin formulation. It includes non linear elasticity in the normal direction as predicted by Hertz for two non-conforming elastic contact bodies. In the shear direction, instead, it reseambles the simplified case without slip discussed in Mindlin's paper, where a linear relationship between shear force and tangential displacement is provided. Finally, the Mohr-Coulomb criterion is employed to established the maximum friction force which can be developed at the contact. Moreover, it is also possible to include the effect of linear viscous damping through the definition of the parameters β_n and β_s.

calcEnergy(*=false*)
: bool to calculate energy terms (shear potential energy, dissipation of energy due to friction and dissipation of energy due to normal and tangential damping)

contactsAdhesive() → float
: Compute total number of adhesive contacts.

frictionDissipation(*=uninitalized*)
: Energy dissipation due to sliding

includeAdhesion(*=false*)
: bool to include the adhesion force following the DMT formulation. If true, also the normal elastic energy takes into account the adhesion effect.

includeMoment(*=false*)
: bool to consider rolling resistance (if Ip2_FrictMat_FrictMat_MindlinPhys::eta is 0.0, no plastic condition is applied.)

neverErase(*=false*)
: Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. Law2_ScGeom_CapillaryPhys_Capillarity)

normDampDissip(*=uninitalized*)
: Energy dissipated by normal damping

normElastEnergy() → float
: Compute normal elastic potential energy. It handles the DMT formulation if Law2_ScGeom_-MindlinPhys_Mindlin::includeAdhesion is set to true.

preventGranularRatcheting(*=true*)
: bool to avoid granular ratcheting

ratioSlidingContacts() → float
: Return the ratio between the number of contacts sliding to the total number at a given time.

shearDampDissip(*=uninitalized*)
: Energy dissipated by tangential damping

shearEnergy(*=uninitalized*)
: Shear elastic potential energy

class yade.wrapper.**Law2_ScGeom_MindlinPhys_MindlinDeresiewitz**(*inherits LawFunctor → Functor → Serializable*)
: Hertz-Mindlin contact law with partial slip solution, as described in [Thornton1991].

neverErase(*=false*)
: Keep interactions even if particles go away from each other (only in case another constitutive law is in the scene, e.g. Law2_ScGeom_CapillaryPhys_Capillarity)

class yade.wrapper.**Law2_ScGeom_ViscElPhys_Basic**(*inherits LawFunctor → Functor → Serializable*)
: Linear viscoelastic model operating on ScGeom and ViscElPhys.

class yade.wrapper.**Law2_ScGeom_WirePhys_WirePM**(*inherits LawFunctor → Functor → Serializable*)
: Constitutive law for the wire model.

linkThresholdIteration(*=1*)
: Iteration to create the link.

7.8.2 LawDispatcher

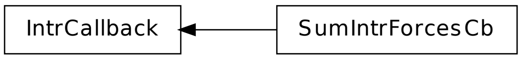
class yade.wrapper.**LawDispatcher**(*inherits Dispatcher → Engine → Serializable*)
: Dispatcher calling functors based on received argument type(s).

dispFunctor(*(IGeom)arg2, (IPhys)arg3*) → LawFunctor
: Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

dispMatrix([*(bool)names=True*]) → dict
: Return dictionary with contents of the dispatch matrix.

functors
: Functors associated with this dispatcher.

7.9 Callbacks



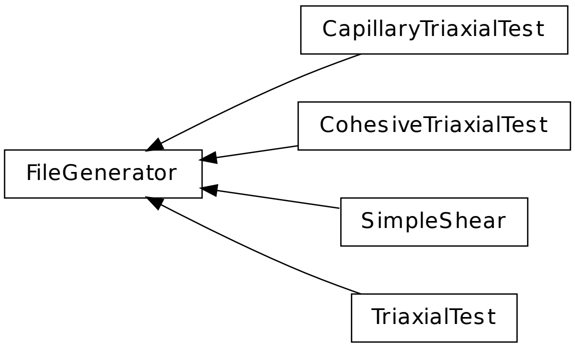
class yade.wrapper.**IntrCallback**(*inherits Serializable*)
: Abstract callback object which will be called for every (real) Interaction after the interaction has been processed by InteractionLoop.

At the beginning of the interaction loop, `stepInit` is called, initializing the object; it returns either `NULL` (to deactivate the callback during this time step) or pointer to function, which will then be passed (1) pointer to the callback object itself and (2) pointer to Interaction.

Note: (NOT YET DONE) This functionality is accessible from python by passing 4th argument to InteractionLoop constructor, or by appending the callback object to InteractionLoop::callbacks.

class yade.wrapper.**SumIntrForcesCb**(*inherits IntrCallback → Serializable*)
Callback summing magnitudes of forces over all interactions. IPhys of interactions must derive from NormShearPhys (responsability fo the user).

7.10 Preprocessors



class yade.wrapper.**FileGenerator**(*inherits Serializable*)
Base class for scene generators, preprocessors.

generate(*(str)out*) → None
Generate scene, save to given file

load() → None
Generate scene, save to temporary file and load immediately

class yade.wrapper.**CapillaryTriaxialTest**(*inherits FileGenerator → Serializable*)
This preprocessor is a variant of TriaxialTest, including the model of capillary forces developed as part of the PhD of Luc Scholtès. See the documentation of Law2_ScGeom_CapillaryPhys_-Capillarity or the main page https://yade-dem.org/wiki/CapillaryTriaxialTest, for more details.

Results obtained with this preprocessor were reported for instance in 'Scholtes et al. Micromechanics of granular materials with capillary effects. International Journal of Engineering Science 2009,(47)1, 64-75.'

CapillaryPressure(*=0*)
Define succion in the packing [Pa]. This is the value used in the capillary model.

Key(*="“*)
A code that is added to output filenames.

Rdispersion(*=0.3*)
Normalized standard deviation of generated sizes.

StabilityCriterion(*=0.01*)
Value of unbalanced force for which the system is considered stable. Used in conditionals to switch between loading stages.

WallStressRecordFile(*=”./WallStressesWater”+Key*)

autoCompressionActivation(*=true*)
Do we just want to generate a stable packing under isotropic pressure (false) or do we want the triaxial loading to start automatically right after compaction stage (true)?

`autoStopSimulation`(*=false*)
: freeze the simulation when conditions are reached (don't activate this if you want to be able to run/stop from Qt GUI)

`autoUnload`(*=true*)
: auto adjust the isotropic stress state from TriaxialTest::sigmaIsoCompaction to TriaxialTest::sigmaLateralConfinement if they have different values. See docs for TriaxialCompressionEngine::autoUnload

`biaxial2dTest`(*=false*)
: FIXME : what is that?

`binaryFusion`(*=true*)
: Defines how overlapping bridges affect the capillary forces (see CapillaryTriaxialTest::fusionDetection). If binary=true, the force is null as soon as there is an overlap detected, if not, the force is divided by the number of overlaps.

`boxFrictionDeg`(*=0.0*)
: Friction angle [°] of boundaries contacts.

`boxKsDivKn`(*=0.5*)
: Ratio of shear vs. normal contact stiffness for boxes.

`boxWalls`(*=true*)
: Use boxes for boundaries (recommended).

`boxYoungModulus`(*=15000000.0*)
: Stiffness of boxes.

`capillaryStressRecordFile`(*=”./capStresses”+Key*)

`compactionFrictionDeg`(*=sphereFrictionDeg*)
: Friction angle [°] of spheres during compaction (different values result in different porosities)]. This value is overridden by TriaxialTest::sphereFrictionDeg before triaxial testing.

`contactStressRecordFile`(*=”./contStresses”+Key*)

`dampingForce`(*=0.2*)
: Coefficient of Cundal-Non-Viscous damping (applied on on the 3 components of forces)

`dampingMomentum`(*=0.2*)
: Coefficient of Cundal-Non-Viscous damping (applied on on the 3 components of torques)

`defaultDt`(*=0.0001*)
: Max time-step. Used as initial value if defined. Latter adjusted by the time stepper.

`density`(*=2600*)
: density of spheres

`facetWalls`(*=false*)
: Use facets for boundaries (not tested)

`finalMaxMultiplier`(*=1.001*)
: max multiplier of diameters during internal compaction (secondary precise adjustment)

`fixedBoxDims`(*=”“*)
: string that contains some subset (max. 2) of {‘x’,’y’,’z’} ; contains axes will have box dimension hardcoded, even if box is scaled as mean_radius is prescribed: scaling will be applied on the rest.

`fixedPoroCompaction`(*=false*)
: flag to choose an isotropic compaction until a fixed porosity choosing a same translation speed for the six walls

`fixedPorosity`(*=1*)
: FIXME : what is that?

`fusionDetection`(*=false*)
: test overlaps between liquid bridges on modify forces if overlaps exist

importFilename(=*”“*)
File with positions and sizes of spheres.

internalCompaction(=*false*)
flag for choosing between moving boundaries or increasing particles sizes during the compaction stage.

lowerCorner(=*Vector3r(0, 0, 0)*)
Lower corner of the box.

maxMultiplier(=*1.01*)
max multiplier of diameters during internal compaction (initial fast increase)

maxWallVelocity(=*10*)
max velocity of boundaries. Usually useless, but can help stabilizing the system in some cases.

noFiles(=*false*)
Do not create any files during run (.xml, .spheres, wall stress records)

numberOfGrains(=*400*)
Number of generated spheres.

radiusControlInterval(=*10*)
interval between size changes when growing spheres.

radiusMean(=*-1*)
Mean radius. If negative (default), autocomputed to as a function of box size and TriaxialTest::numberOfGrains

recordIntervalIter(=*20*)
interval between file outputs

sigmaIsoCompaction(=*50000*)
Confining stress during isotropic compaction.

sigmaLateralConfinement(=*50000*)
Lateral stress during triaxial loading. An isotropic unloading is performed if the value is not equal to CapillaryTriaxialTest::SigmaIsoCompaction.

sphereFrictionDeg(=*18.0*)
Friction angle [°] of spheres assigned just before triaxial testing.

sphereKsDivKn(=*0.5*)
Ratio of shear vs. normal contact stiffness for spheres.

sphereYoungModulus(=*15000000.0*)
Stiffness of spheres.

strainRate(=*1*)
Strain rate in triaxial loading.

thickness(=*0.001*)
thickness of boundaries. It is arbitrary and should have no effect

timeStepOutputInterval(=*50*)
interval for outputing general information on the simulation (stress,unbalanced force,...)

timeStepUpdateInterval(=*50*)
interval for GlobalStiffnessTimeStepper

upperCorner(=*Vector3r(1, 1, 1)*)
Upper corner of the box.

wallOversizeFactor(=*1.3*)
Make boundaries larger than the packing to make sure spheres don’t go out during deformation.

wallStiffnessUpdateInterval(=*10*)
interval for updating the stiffness of sample/boundaries contacts

wallWalls(*=false*)
: Use walls for boundaries (not tested)

water(*=true*)
: activate capillary model

class yade.wrapper.**CohesiveTriaxialTest**(*inherits FileGenerator → Serializable*)

This preprocessor is a variant of TriaxialTest using the cohesive-frictional contact law with moments. It sets up a scene for cohesive triaxial tests. See full documentation at http://yade-dem.org/wiki/TriaxialTest.

Cohesion is initially 0 by default. The suggested usage is to define cohesion values in a second step, after isotropic compaction : define shear and normal cohesions in Ip2_CohFrictMat_CohFrictMat_CohFrictPhys, then turn Ip2_CohFrictMat_CohFrictMat_CohFrictPhys::setCohesionNow true to assign them at each contact at next iteration.

Key(*=""*)
: A code that is added to output filenames.

StabilityCriterion(*=0.01*)
: Value of unbalanced force for which the system is considered stable. Used in conditionals to switch between loading stages.

WallStressRecordFile(*="./CohesiveWallStresses"+Key*)

autoCompressionActivation(*=true*)
: Do we just want to generate a stable packing under isotropic pressure (false) or do we want the triaxial loading to start automatically right after compaction stage (true)?

autoStopSimulation(*=false*)
: freeze the simulation when conditions are reached (don't activate this if you want to be able to run/stop from Qt GUI)

autoUnload(*=true*)
: auto adjust the isotropic stress state from TriaxialTest::sigmaIsoCompaction to TriaxialTest::sigmaLateralConfinement if they have different values. See docs for TriaxialCompressionEngine::autoUnload

biaxial2dTest(*=false*)
: FIXME : what is that?

boxFrictionDeg(*=0.0*)
: Friction angle [°] of boundaries contacts.

boxKsDivKn(*=0.5*)
: Ratio of shear vs. normal contact stiffness for boxes.

boxWalls(*=true*)
: Use boxes for boundaries (recommended).

boxYoungModulus(*=15000000.0*)
: Stiffness of boxes.

compactionFrictionDeg(*=sphereFrictionDeg*)
: Friction angle [°] of spheres during compaction (different values result in different porosities)]. This value is overridden by TriaxialTest::sphereFrictionDeg before triaxial testing.

dampingForce(*=0.2*)
: Coefficient of Cundal-Non-Viscous damping (applied on on the 3 components of forces)

dampingMomentum(*=0.2*)
: Coefficient of Cundal-Non-Viscous damping (applied on on the 3 components of torques)

defaultDt(*=0.001*)
: Max time-step. Used as initial value if defined. Latter adjusted by the time stepper.

density(*=2600*)
: density of spheres

`facetWalls`(*=false*)
: Use facets for boundaries (not tested)

`finalMaxMultiplier`(*=1.001*)
: max multiplier of diameters during internal compaction (secondary precise adjustment)

`fixedBoxDims`(*=”“*)
: string that contains some subset (max. 2) of {‘x’,’y’,’z’} ; contains axes will have box dimension hardcoded, even if box is scaled as mean_radius is prescribed: scaling will be applied on the rest.

`fixedPoroCompaction`(*=false*)
: flag to choose an isotropic compaction until a fixed porosity choosing a same translation speed for the six walls

`fixedPorosity`(*=1*)
: FIXME : what is that?

`importFilename`(*=”“*)
: File with positions and sizes of spheres.

`internalCompaction`(*=false*)
: flag for choosing between moving boundaries or increasing particles sizes during the compaction stage.

`lowerCorner`(*=Vector3r(0, 0, 0)*)
: Lower corner of the box.

`maxMultiplier`(*=1.01*)
: max multiplier of diameters during internal compaction (initial fast increase)

`maxWallVelocity`(*=10*)
: max velocity of boundaries. Usually useless, but can help stabilizing the system in some cases.

`noFiles`(*=false*)
: Do not create any files during run (.xml, .spheres, wall stress records)

`normalCohesion`(*=0*)
: Material parameter used to define contact strength in tension.

`numberOfGrains`(*=400*)
: Number of generated spheres.

`radiusControlInterval`(*=10*)
: interval between size changes when growing spheres.

`radiusDeviation`(*=0.3*)
: Normalized standard deviation of generated sizes.

`radiusMean`(*=-1*)
: Mean radius. If negative (default), autocomputed to as a function of box size and TriaxialTest::numberOfGrains

`recordIntervalIter`(*=20*)
: interval between file outputs

`setCohesionOnNewContacts`(*=false*)
: create cohesionless (False) or cohesive (True) interactions for new contacts.

`shearCohesion`(*=0*)
: Material parameter used to define shear strength of contacts.

`sigmaIsoCompaction`(*=50000*)
: Confining stress during isotropic compaction.

`sigmaLateralConfinement`(*=50000*)
: Lateral stress during triaxial loading. An isotropic unloading is performed if the value is not equal to TriaxialTest::sigmaIsoCompaction.

sphereFrictionDeg(*=18.0*)
Friction angle [°] of spheres assigned just before triaxial testing.

sphereKsDivKn(*=0.5*)
Ratio of shear vs. normal contact stiffness for spheres.

sphereYoungModulus(*=15000000.0*)
Stiffness of spheres.

strainRate(*=0.1*)
Strain rate in triaxial loading.

thickness(*=0.001*)
thickness of boundaries. It is arbitrary and should have no effect

timeStepUpdateInterval(*=50*)
interval for GlobalStiffnessTimeStepper

upperCorner(*=Vector3r(1, 1, 1)*)
Upper corner of the box.

wallOversizeFactor(*=1.3*)
Make boundaries larger than the packing to make sure spheres don't go out during deformation.

wallStiffnessUpdateInterval(*=10*)
interval for updating the stiffness of sample/boundaries contacts

wallWalls(*=false*)
Use walls for boundaries (not tested)

class yade.wrapper.**SimpleShear**(*inherits FileGenerator → Serializable*)
Preprocessor for creating a numerical model of a simple shear box.

- Boxes (6) constitute the different sides of the box itself
- Spheres are contained in the box. The sample is generated by default via the same method used in TriaxialTest Preprocesor (=> see in source function GenerateCloud). But import of a list of spheres from a text file can be also performed after few changes in the source code.

  Launching this preprocessor will carry out an oedometric compression, until a value of normal stress equal to 2 MPa (and stable). But with others Engines KinemCNDEngine, KinemCNSEngine and KinemCNLEngine, respectively constant normal displacement, constant normal rigidity and constant normal stress paths can be carried out for such simple shear boxes.

NB about micro-parameters : their default values correspond to those used in [Duriez2009a] and [Duriez2011] to simulate infilled rock joints.

boxPoissonRatio(*=0.04*)
value of ElastMat::poisson for the spheres [-]

boxYoungModulus(*=4.0e9*)
value of ElastMat::young for the boxes [Pa]

density(*=2600*)
density of the spheres [$\mathrm{kg/m^3}$]

gravApplied(*=false*)
depending on this, GravityEngine is added or not to the scene to take into account the weight of particles

gravity(*=Vector3r(0, -9.81, 0)*)
vector corresponding to used gravity (if :yref:gravApplied<SimpleShear::gravApplied>‘) [$\mathrm{m/s^2}$]

height(*=0.02*)
initial height (along y-axis) of the shear box [m]

length(*=0.1*)
: initial length (along x-axis) of the shear box [m]

sphereFrictionDeg(*=37*)
: value of ElastMat::poisson for the spheres [°] (the necessary conversion in rad is done automatically)

spherePoissonRatio(*=0.04*)
: value of ElastMat::poisson for the spheres [-]

sphereYoungModulus(*=4.0e9*)
: value of ElastMat::young for the spheres [Pa]

thickness(*=0.001*)
: thickness of the boxes constituting the shear box [m]

timeStepUpdateInterval(*=50*)
: value of TimeStepper::timeStepUpdateInterval for the TimeStepper used here

width(*=0.04*)
: initial width (along z-axis) of the shear box [m]

class `yade.wrapper.`**TriaxialTest**(*inherits FileGenerator → Serializable*)
Create a scene for triaxal test.

Introduction Yade includes tools to simulate triaxial tests on particles assemblies. This preprocessor (and variants like e.g. CapillaryTriaxialTest) illustrate how to use them. It generates a scene which will - by default - go through the following steps :

- generate random loose packings in a parallelepiped.
- compress the packing isotropicaly, either squeezing the packing between moving rigid boxes or expanding the particles while boxes are fixed (depending on flag internalCompaction). The confining pressure in this stage is defined via sigmaIsoCompaction.
- when the packing is dense and stable, simulate a loading path and get the mechanical response as a result.

The default loading path corresponds to a constant lateral stress (sigmaLateralConfinement) in 2 directions and constant strain rate on the third direction. This default loading path is performed when the flag autoCompressionActivation it `True`, otherwise the simulation stops after isotropic compression.

Different loading paths might be performed. In order to define them, the user can modify the flags found in engine TriaxialStressController at any point in the simulation (in c++). If `TriaxialStressController.wall_X_activated` is `true` boundary X is moved automatically to maintain the defined stress level *sigmaN* (see axis conventions below). If `false` the boundary is not controlled by the engine at all. In that case the user is free to prescribe fixed position, constant velocity, or more complex conditions.

Note: *Axis conventions.* Boundaries perpendicular to the x axis are called "left" and "right", y corresponds to "top" and "bottom", and axis z to "front" and "back". In the default loading path, strain rate is assigned along y, and constant stresses are assigned on x and z.

Essential engines

1. The TrixaialCompressionEngine is used for controlling the state of the sample and simulating loading paths. TriaxialCompressionEngine inherits from TriaxialStressController, which computes stress- and strain-like quantities in the packing and maintain a constant level of stress at each boundary. TriaxialCompressionEngine has few more members in order to impose constant strain rate and control the transition between isotropic compression and triaxial test. Transitions are defined by changing some flags of the TriaxialStressController, switching from/to imposed strain rate to/from imposed stress.
2. The class TriaxialStateRecorder is used to write to a file the history of stresses and strains.

3. TriaxialTest is using GlobalStiffnessTimeStepper to compute an appropriate Δt for the numerical scheme.

Note: `TriaxialStressController::ComputeUnbalancedForce` returns a value that can be useful for evaluating the stability of the packing. It is defined as (mean force on particles)/(mean contact force), so that it tends to 0 in a stable packing. This parameter is checked by TriaxialCompressionEngine to switch from one stage of the simulation to the next one (e.g. stop isotropic confinment and start axial loading)

Frequently Asked Questions

1. **How is generated the packing? How to change particles sizes distribution? Why do I have a m**
   The initial positioning of spheres is done by generating random (x,y,z) in a box and checking if a sphere of radius R (R also randomly generated with respect to a uniform distribution between mean*(1-std_dev) and mean*(1+std_dev) can be inserted at this location without overlaping with others.

   If the sphere overlaps, new (x,y,z)'s are generated until a free position for the new sphere is found. This explains the message you have: after 3000 trial-and-error, the sphere couldn't be placed, and the algorithm stops.

   You get the message above if you try to generate an initialy dense packing, which is not possible with this algorithm. It can only generate clouds. You should keep the default value of porosity (n~0.7), or even increase if it is still to low in some cases. The dense state will be obtained in the second step (compaction, see below).

2. **How is the compaction done, what are the parameters maxWallVelocity and finalMaxMultiplie**

   Compaction is done

   (a) by moving rigid boxes or

   (b) by increasing the sizes of the particles (decided using the option internalCompaction size increase).

   Both algorithm needs numerical parameters to prevent instabilities. For instance, with the method (1) maxWallVelocity is the maximum wall velocity, with method (2) finalMaxMultiplier is the max value of the multiplier applied on sizes at each iteration (always something like 1.00001).

3. **During the simulation of triaxial compression test, the wall in one direction moves with an inc**
   The control of stress on a boundary is based on the total stiffness K of all contacts between the packing and this boundary. In short, at each step, displacement=stress_error/K. This algorithm is implemented in TriaxialStressController, and the control itself is in `TriaxialStressController::ControlExternalStress`. The control can be turned off independently for each boundary, using the flags `wall_XXX_activated`, with *XXX* {*top*, *bottom*, *left*, *right*, *back*, *front*}. The imposed sress is a unique value (sigma_iso) for all directions if TriaxialStressController.isAxisymetric, or 3 independent values sigma1, sigma2, sigma3.

4. **Which value of friction angle do you use during the compaction phase of the Triaxial Test?**
   The friction during the compaction (whether you are using the expansion method or the compression one for the specimen generation) can be anything between 0 and the final value used during the Triaxial phase. Note that higher friction than the final one would result in volumetric collapse at the beginning of the test. The purpose of using a different value of friction during this phase is related to the fact that the final porosity you get at the end of the sample generation essentially depends on it as well as on the assumed Particle Size Distribution. Changing the initial value of friction will get to a different value of the final porosity.

5. **Which is the aim of the `bool isRadiusControlIteration`?** This internal variable (updated automatically) is true each *N* timesteps (with *N*=radiusControlInterval). For other

timesteps, there is no expansion. Cycling without expanding is just a way to speed up the simulation, based on the idea that 1% increase each 10 iterations needs less operations than 0.1% at each iteration, but will give similar results.

6. **How comes the unbalanced force reaches a low value only after many timesteps in the compact**
The value of unbalanced force (dimensionless) is expected to reach low value (i.e. identifying a static-equilibrium condition for the specimen) only at the end of the compaction phase. The code is not aiming at simulating a quasistatic isotropic compaction process, it is only giving a stable packing at the end of it.

`Key`(*=”“*)
A code that is added to output filenames.

`StabilityCriterion`(*=0.01*)
Value of unbalanced force for which the system is considered stable. Used in conditionals to switch between loading stages.

`WallStressRecordFile`(*=”./WallStresses”+Key*)

`autoCompressionActivation`(*=true*)
Do we just want to generate a stable packing under isotropic pressure (false) or do we want the triaxial loading to start automatically right after compaction stage (true)?

`autoStopSimulation`(*=false*)
freeze the simulation when conditions are reached (don’t activate this if you want to be able to run/stop from Qt GUI)

`autoUnload`(*=true*)
auto adjust the isotropic stress state from TriaxialTest::sigmaIsoCompaction to TriaxialTest::sigmaLateralConfinement if they have different values. See docs for TriaxialCompressionEngine::autoUnload

`biaxial2dTest`(*=false*)
FIXME : what is that?

`boxFrictionDeg`(*=0.0*)
Friction angle [°] of boundaries contacts.

`boxKsDivKn`(*=0.5*)
Ratio of shear vs. normal contact stiffness for boxes.

`boxYoungModulus`(*=15000000.0*)
Stiffness of boxes.

`compactionFrictionDeg`(*=sphereFrictionDeg*)
Friction angle [°] of spheres during compaction (different values result in different porosities)]. This value is overridden by TriaxialTest::sphereFrictionDeg before triaxial testing.

`dampingForce`(*=0.2*)
Coefficient of Cundal-Non-Viscous damping (applied on on the 3 components of forces)

`dampingMomentum`(*=0.2*)
Coefficient of Cundal-Non-Viscous damping (applied on on the 3 components of torques)

`defaultDt`(*=-1*)
Max time-step. Used as initial value if defined. Latter adjusted by the time stepper.

`density`(*=2600*)
density of spheres

`facetWalls`(*=false*)
Use facets for boundaries (not tested)

`finalMaxMultiplier`(*=1.001*)
max multiplier of diameters during internal compaction (secondary precise adjustment)

`fixedBoxDims`(*=”“*)
string that contains some subset (max. 2) of {‘x’,’y’,’z’} ; contains axes will have box dimension

hardcoded, even if box is scaled as mean_radius is prescribed: scaling will be applied on the rest.

`importFilename`(*=""*)
File with positions and sizes of spheres.

`internalCompaction`(*=false*)
flag for choosing between moving boundaries or increasing particles sizes during the compaction stage.

`lowerCorner`(*=Vector3r(0, 0, 0)*)
Lower corner of the box.

`maxMultiplier`(*=1.01*)
max multiplier of diameters during internal compaction (initial fast increase)

`maxWallVelocity`(*=10*)
max velocity of boundaries. Usually useless, but can help stabilizing the system in some cases.

`noFiles`(*=false*)
Do not create any files during run (.xml, .spheres, wall stress records)

`numberOfGrains`(*=400*)
Number of generated spheres.

`radiusControlInterval`(*=10*)
interval between size changes when growing spheres.

`radiusMean`(*=-1*)
Mean radius. If negative (default), autocomputed to as a function of box size and TriaxialTest::numberOfGrains

`radiusStdDev`(*=0.3*)
Normalized standard deviation of generated sizes.

`recordIntervalIter`(*=20*)
interval between file outputs

`sigmaIsoCompaction`(*=50000*)
Confining stress during isotropic compaction.

`sigmaLateralConfinement`(*=50000*)
Lateral stress during triaxial loading. An isotropic unloading is performed if the value is not equal to TriaxialTest::sigmaIsoCompaction.

`sphereFrictionDeg`(*=18.0*)
Friction angle [°] of spheres assigned just before triaxial testing.

`sphereKsDivKn`(*=0.5*)
Ratio of shear vs. normal contact stiffness for spheres.

`sphereYoungModulus`(*=15000000.0*)
Stiffness of spheres.

`strainRate`(*=0.1*)
Strain rate in triaxial loading.

`thickness`(*=0.001*)
thickness of boundaries. It is arbitrary and should have no effect

`timeStepUpdateInterval`(*=50*)
interval for GlobalStiffnessTimeStepper

`upperCorner`(*=Vector3r(1, 1, 1)*)
Upper corner of the box.

`wallOversizeFactor`(*=1.3*)
Make boundaries larger than the packing to make sure spheres don't go out during deformation.

wallStiffnessUpdateInterval(*=10*)
: interval for updating the stiffness of sample/boundaries contacts

wallWalls(*=false*)
: Use walls for boundaries (not tested)

7.11 Rendering

7.11.1 OpenGLRenderer

class `yade.wrapper.`**OpenGLRenderer**(*inherits Serializable*)
: Class responsible for rendering scene on OpenGL devices.

bgColor(*=Vector3r(.2, .2, .2)*)
: Color of the background canvas (RGB)

bound(*=false*)
: Render body Bound

clipPlaneActive(*=vector<bool>(numClipPlanes, false)*)
: Activate/deactivate respective clipping planes

clipPlaneSe3(*=vector<Se3r>(numClipPlanes, Se3r(Vector3r::Zero(), Quaternionr::Identity()))*)
: Position and orientation of clipping planes

dispScale(*=Vector3r::Ones(), disable scaling*)
: Artificially enlarge (scale) dispalcements from bodies' reference positions by this relative amount, so that they become better visible (independently in 3 dimensions). Disbled if (1,1,1).

dof(*=false*)
: Show which degrees of freedom are blocked for each body

extraDrawers(*=uninitalized*)
: Additional rendering components (GlExtraDrawer).

ghosts(*=true*)
: Render objects crossing periodic cell edges by cloning them in multiple places (periodic simulations only).

id(*=false*)
: Show body id's

intrAllWire(*=false*)
: Draw wire for all interactions, blue for potential and green for real ones (mostly for debugging)

intrGeom(*=false*)
: Render Interaction::geom objects.

intrPhys(*=false*)
: Render Interaction::phys objects

intrWire(*=false*)
: If rendering interactions, use only wires to represent them.

light1(*=true*)
: Turn light 1 on.

light2(*=true*)
: Turn light 2 on.

light2Color(*=Vector3r(0.5, 0.5, 0.1)*)
: Per-color intensity of secondary light (RGB).

light2Pos(*=Vector3r(-130, 75, 30)*)
: Position of secondary OpenGL light source in the scene.

lightColor(*=Vector3r(0.6, 0.6, 0.6)*)
Per-color intensity of primary light (RGB).

lightPos(*=Vector3r(75, 130, 0)*)
Position of OpenGL light source in the scene.

mask(*=~0, draw everything*)
Bitmask for showing only bodies where ((mask & Body::mask)!=0)

render() → None
Render the scene in the current OpenGL context.

rotScale(*=1., disable scaling*)
Artificially enlarge (scale) rotations of bodies relative to their reference orientation, so the they are better visible.

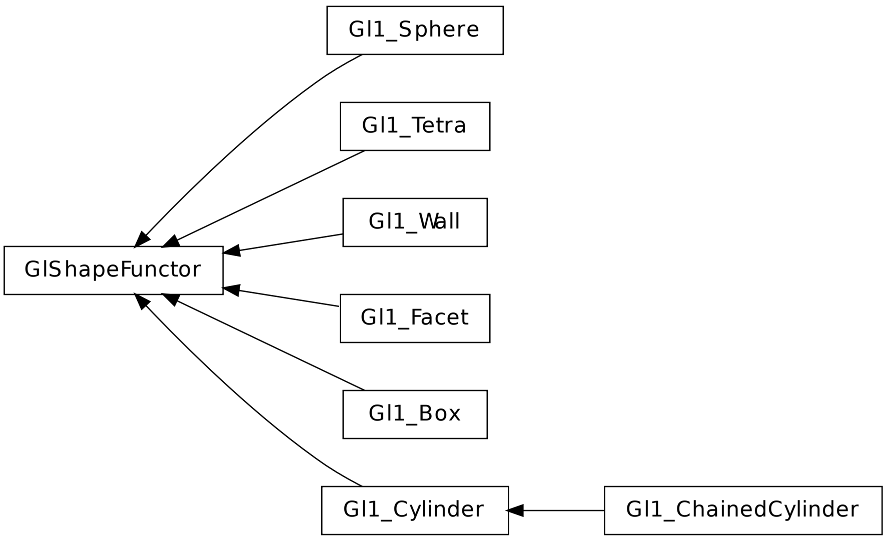
selId(*=Body::ID_NONE*)
Id of particle that was selected by the user.

setRefSe3() → None
Make current positions and orientation reference for scaleDisplacements and scaleRotations.

shape(*=true*)
Render body Shape

wire(*=false*)
Render all bodies with wire only (faster)

7.11.2 GlShapeFunctor



class yade.wrapper.**GlShapeFunctor**(*inherits Functor → Serializable*)
Abstract functor for rendering Shape objects.

class yade.wrapper.**Gl1_Box**(*inherits GlShapeFunctor → Functor → Serializable*)
Renders Box object

class yade.wrapper.**Gl1_ChainedCylinder**(*inherits Gl1_Cylinder → GlShapeFunctor → Functor → Serializable*)
Renders ChainedCylinder object including a shift for compensating flexion.

class yade.wrapper.**Gl1_Cylinder**(*inherits GlShapeFunctor → Functor → Serializable*)
Renders Cylinder object

wire(=*false*) [**static**]
: Only show wireframe (controlled by `glutSlices` and `glutStacks`.

glutNormalize(=*true*) [**static**]
: Fix normals for non-wire rendering

glutSlices(=*8*) [**static**]
: Number of sphere slices.

glutStacks(=*4*) [**static**]
: Number of sphere stacks.

glutNormalize = **True**

glutSlices = **8**

glutStacks = **4**

wire = **False**

class yade.wrapper.**Gl1_Facet**(*inherits GlShapeFunctor → Functor → Serializable*)
: Renders Facet object

normals(=*false*) [**static**]
: In wire mode, render normals of facets and edges; facet's colors are disregarded in that case.

normals = **False**

class yade.wrapper.**Gl1_Sphere**(*inherits GlShapeFunctor → Functor → Serializable*)
: Renders Sphere object

quality(=*1.0*) [**static**]
: Change discretization level of spheres. quality>1 for better image quality, at the price of more cpu/gpu usage, 0<quality<1 for faster rendering. If mono-color sphres are displayed (Gl1_Sphere::stripes=False), quality mutiplies :yref:'Gl1_Sphere::glutSlices and Gl1_Sphere::glutStacks. If striped spheres are displayed (:yref:'Gl1_Sphere::stripes=True), only integer increments are meaningfull : quality=1 and quality=1.9 will give the same result, quality=2 will give finer result.

wire(=*false*) [**static**]
: Only show wireframe (controlled by `glutSlices` and `glutStacks`.

stripes(=*false*) [**static**]
: In non-wire rendering, show stripes clearly showing particle rotation.

localSpecView(=*true*) [**static**]
: Compute specular light in local eye coordinate system.

glutSlices(=*12*) [**static**]
: Base number of sphere slices, multiplied by Gl1_Sphere::quality before use); not used with `stripes` (see glut{Solid,Wire}Sphere reference)

glutStacks(=*6*) [**static**]
: Base number of sphere stacks, multiplied by Gl1_Sphere::quality before use; not used with `stripes` (see glut{Solid,Wire}Sphere reference)

glutSlices = **12**

glutStacks = **6**

localSpecView = **True**

quality = **1.0**

stripes = **False**

wire = **False**

class yade.wrapper.**Gl1_Tetra**(*inherits GlShapeFunctor → Functor → Serializable*)
: Renders Tetra object

class yade.wrapper.**Gl1_Wall**(*inherits GlShapeFunctor → Functor → Serializable*)
: Renders Wall object

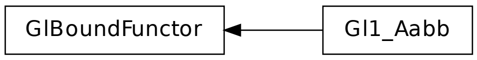
`div`(*=20*) [**static**]
Number of divisions of the wall inside visible scene part.

`div` = **20**

7.11.3 GlStateFunctor

class `yade.wrapper.GlStateFunctor`(*inherits Functor → Serializable*)
Abstract functor for rendering State objects.

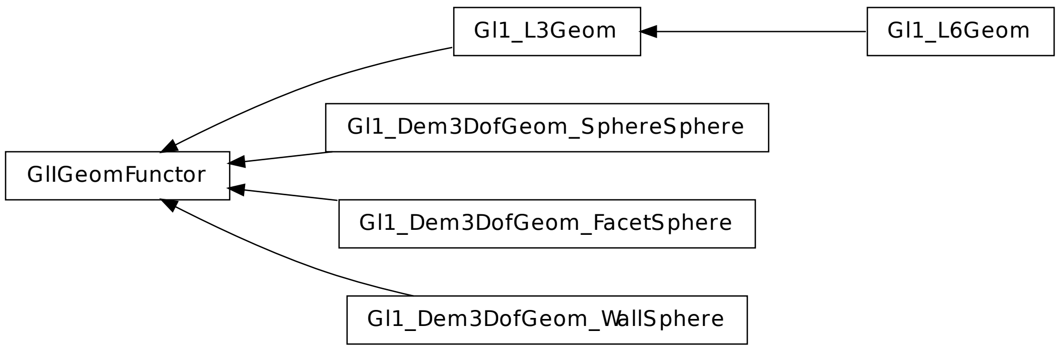
7.11.4 GlBoundFunctor



class `yade.wrapper.GlBoundFunctor`(*inherits Functor → Serializable*)
Abstract functor for rendering Bound objects.

class `yade.wrapper.Gl1_Aabb`(*inherits GlBoundFunctor → Functor → Serializable*)
Render Axis-aligned bounding box (Aabb).

7.11.5 GlIGeomFunctor


Gl1_L3Geom
Gl1_L6Geom
Gl1_Dem3DofGeom_SphereSphere
GlIGeomFunctor
Gl1_Dem3DofGeom_FacetSphere
Gl1_Dem3DofGeom_WallSphere

class `yade.wrapper.GlIGeomFunctor`(*inherits Functor → Serializable*)
Abstract functor for rendering IGeom objects.

class `yade.wrapper.Gl1_Dem3DofGeom_FacetSphere`(*inherits GlIGeomFunctor → Functor → Serializable*)
Render interaction of facet and sphere (represented by Dem3DofGeom_FacetSphere)

`normal`(*=false*) [**static**]
Render interaction normal

`rolledPoints`(*=false*) [**static**]
Render points rolled on the sphere & facet (original contact point)

`unrolledPoints`(*=false*) [**static**]
Render original contact points unrolled to the contact plane

`shear`(*=false*) [**static**]
Render shear line in the contact plane

`shearLabel`(*=false*) [**static**]
Render shear magnitude as number

`normal` = **False**

`rolledPoints` = **False**

`shear` = **False**

`shearLabel` = **False**

`unrolledPoints` = **False**

class `yade.wrapper.Gl1_Dem3DofGeom_SphereSphere`(*inherits GlIGeomFunctor → Functor → Serializable*)
Render interaction of 2 spheres (represented by Dem3DofGeom_SphereSphere)

`normal`(*=false*) [**static**]
Render interaction normal

`rolledPoints`(*=false*) [**static**]
Render points rolled on the spheres (tracks the original contact point)

`unrolledPoints`(*=false*) [**static**]
Render original contact points unrolled to the contact plane

`shear`(*=false*) [**static**]
Render shear line in the contact plane

`shearLabel`(*=false*) [**static**]
Render shear magnitude as number

`normal` = **False**

`rolledPoints` = **False**

`shear` = **False**

`shearLabel` = **False**

`unrolledPoints` = **False**

class `yade.wrapper.Gl1_Dem3DofGeom_WallSphere`(*inherits GlIGeomFunctor → Functor → Serializable*)
Render interaction of wall and sphere (represented by Dem3DofGeom_WallSphere)

`normal`(*=false*) [**static**]
Render interaction normal

`rolledPoints`(*=false*) [**static**]
Render points rolled on the spheres (tracks the original contact point)

`unrolledPoints`(*=false*) [**static**]
Render original contact points unrolled to the contact plane

`shear`(*=false*) [**static**]
Render shear line in the contact plane

`shearLabel`(*=false*) [**static**]
Render shear magnitude as number

`normal` = **False**

`rolledPoints` = **False**

`shear` = **False**

`shearLabel` = **False**

`unrolledPoints` = **False**

class `yade.wrapper.Gl1_L3Geom`(*inherits GlIGeomFunctor → Functor → Serializable*)
Render L3Geom geometry.

`axesLabels`(*=false*) [**static**]
Whether to display labels for local axes (x,y,z)

`axesScale`(*=1.*) **[static]**
: Scale local axes, their reference length being half of the minimum radius.

`axesWd`(*=1.*) **[static]**
: Width of axes lines, in pixels; not drawn if non-positive

`uPhiWd`(*=2.*) **[static]**
: Width of lines for drawing displacements (and rotations for L6Geom); not drawn if non-positive.

`uScale`(*=1.*) **[static]**
: Scale local displacements (u - u0); 1 means the true scale, 0 disables drawing local displacements; negative values are permissible.

`axesLabels` **= False**

`axesScale` **= 1.0**

`axesWd` **= 1.0**

`uPhiWd` **= 2.0**

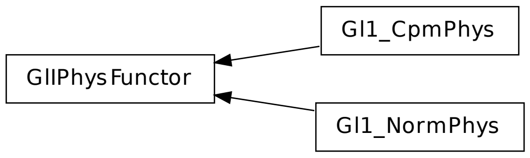
`uScale` **= 1.0**

class `yade.wrapper.Gl1_L6Geom`(*inherits Gl1_L3Geom → GlIGeomFunctor → Functor → Serializable*)
: Render L6Geom geometry.

`phiScale`(*=1.*) **[static]**
: Scale local rotations (phi - phi0). The default scale is to draw π rotation with length equal to minimum radius.

`phiScale` **= 1.0**

7.11.6 GlIPhysFunctor



class `yade.wrapper.GlIPhysFunctor`(*inherits Functor → Serializable*)
: Abstract functor for rendering IPhys objects.

class `yade.wrapper.Gl1_CpmPhys`(*inherits GlIPhysFunctor → Functor → Serializable*)
: Render CpmPhys objects of interactions.

`contactLine`(*=true*) **[static]**
: Show contact line

`dmgLabel`(*=true*) **[static]**
: Numerically show contact damage parameter

`dmgPlane`(*=false*) **[static]**
: [what is this?]

`epsT`(*=false*) **[static]**
: Show shear strain

`epsTAxes`(*=false*) **[static]**
: Show axes of shear plane

`normal`(*=false*) **[static]**
: Show contact normal

colorStrainRatio(*=-1*) **[static]**
If positive, set the interaction (wire) color based on ε_N normalized by ε_0 × *colorStrainRatio* (ε_0=:yref:*CpmPhys.epsCrackOnset*). Otherwise, color based on the residual strength.

epsNLabel(*=false*) **[static]**
Numerically show normal strain

colorStrainRatio = -1.0

contactLine = True

dmgLabel = True

dmgPlane = False

epsNLabel = False

epsT = False

epsTAxes = False

normal = False

class yade.wrapper.Gl1_NormPhys(*inherits GlIPhysFunctor → Functor → Serializable*)
Renders NormPhys objects as cylinders of which diameter and color depends on NormPhys:normForce magnitude.

maxFn(*=0*) **[static]**
Value of NormPhys.normalForce corresponding to maxDiameter. This value will be increased (but *not decreased*) automatically.

signFilter(*=0*) **[static]**
If non-zero, only display contacts with negative (-1) or positive (+1) normal forces; if zero, all contacts will be displayed.

refRadius(*=std::numeric_limits<Real>::infinity()*) **[static]**
Reference (minimum) particle radius; used only if maxRadius is negative. This value will be decreased (but *not increased*) automatically. *(auto-updated)*

maxRadius(*=-1*) **[static]**
Cylinder radius corresponding to the maximum normal force. If negative, auto-updated refRadius will be used instead.

slices(*=6*) **[static]**
Number of sphere slices; (see glutCylinder reference)

stacks(*=1*) **[static]**
Number of sphere stacks; (see glutCylinder reference)

maxWeakFn(*=NaN*) **[static]**
Value that divides contacts by their normal force into the ‘‘weak fabric” and ‘‘strong fabric’‘. This value is set as side-effect by utils.fabricTensor.

weakFilter(*=0*) **[static]**
If non-zero, only display contacts belonging to the ‘‘weak” (-1) or ‘‘strong” (+1) fabric.

weakScale(*=1.*) **[static]**
If maxWeakFn is set, scale radius of the weak fabric by this amount (usually smaller than 1). If zero, 1 pixel line is displayed. Colors are not affected by this value.

maxFn = 0.0

maxRadius = -1.0

maxWeakFn = nan

refRadius = inf

signFilter = 0

slices = 6

stacks = 1

`weakFilter` = 0

`weakScale` = 1.0

7.12 Simulation data

7.12.1 Omega

class `yade.wrapper.Omega`

`addScene()` → int
: Add new scene to Omega, returns its number

`bodies`
: Bodies in the current simulation (container supporting index access by id and iteration)

`cell`
: Periodic cell of the current scene (None if the scene is aperiodic).

`childClassesNonrecursive(`*(str)arg2*`)` → list
: Return list of all classes deriving from given class, as registered in the class factory

`disableGdb()` → None
: Revert SEGV and ABRT handlers to system defaults.

`dt`
: Current timestep (Δt) value.

  - assigning negative value enables dynamic Δt (by looking for a TimeStepper in O.engine) and sets positive timestep `O.dt=|Δt|` (will be used until the timestepper is run and updates it)
  - assigning positive value sets Δt to that value and disables dynamic Δt (via TimeStepper, if there is one).

  dynDt can be used to query whether dynamic Δt is in use.

`dynDt`
: Whether a TimeStepper is used for dynamic Δt control. See dt on how to enable/disable TimeStepper.

`dynDtAvailable`
: Whether a TimeStepper is amongst O.engines, activated or not.

`energy`
: EnergyTracker of the current simulation. (meaningful only with O.trackEnergy)

`engines`
: List of engines in the simulation (Scene::engines).

`exitNoBacktrace(`[*(int)status=0*]`)` → None
: Disable SEGV handler and exit, optionally with given status number.

`filename`
: Filename under which the current simulation was saved (None if never saved).

`forceSyncCount`
: Counter for number of syncs in ForceContainer, for profiling purposes.

`forces`
: ForceContainer (forces, torques, displacements) in the current simulation.

`interactions`
: Interactions in the current simulation (container supporting index acces by either (id1,id2) or interactionNumber and iteration)

isChildClassOf(*(str)arg2*, *(str)arg3*) → bool
: Tells whether the first class derives from the second one (both given as strings).

iter
: Get current step number

labeledEngine(*(str)arg2*) → object
: Return instance of engine/functor with the given label. This function shouldn't be called by the user directly; every ehange in O.engines will assign respective global python variables according to labels.

  For example:: O.engines=[InsertionSortCollider(label='collider')] collider.nBins=5 ## collider has become a variable after assignment to O.engines automatically)

load(*(str)file*[, *(bool)quiet=False*]) → None
: Load simulation from file.

loadTmp([*(str)mark=''*[, *(bool)quiet=False*]]) → None
: Load simulation previously stored in memory by saveTmp. *mark* optionally distinguishes multiple saved simulations

lsTmp() → list
: Return list of all memory-saved simulations.

materials
: Shared materials; they can be accessed by id or by label

miscParams
: MiscParams in the simulation (Scene::mistParams), usually used to save serializables that don't fit anywhere else, like GL functors

numThreads
: Get maximum number of threads openMP can use.

pause() → None
: Stop simulation execution. (May be called from within the loop, and it will stop after the current step).

periodic
: Get/set whether the scene is periodic or not (True/False).

plugins() → list
: Return list of all plugins registered in the class factory.

realtime
: Return clock (human world) time the simulation has been running.

reload([*(bool)quiet=False*]) → None
: Reload current simulation

reset() → None
: Reset simulations completely (including another scenes!).

resetAllScenes() → None
: Reset all scenes.

resetCurrentScene() → None
: Reset current scene.

resetThisScene() → None
: Reset current scene.

resetTime() → None
: Reset simulation time: step number, virtual and real time. (Doesn't touch anything else, including timings).

run([*(int)nSteps=-1*[, *(bool)wait=False*]]) → None
: Run the simulation. *nSteps* how many steps to run, then stop (if positive); *wait* will cause not returning to python until simulation will have stopped.

`runEngine`(*(Engine)arg2*) → None
: Run given engine exactly once; simulation time, step number etc. will not be incremented (use only if you know what you do).

`running`
: Whether background thread is currently running a simulation.

`save`(*(str)file*[, *(bool)quiet=False*]) → None
: Save current simulation to file (should be .xml or .xml.bz2)

`saveTmp`([*(str)mark=''*[, *(bool)quiet=False*]]) → None
: Save simulation to memory (disappears at shutdown), can be loaded later with loadTmp. *mark* optionally distinguishes different memory-saved simulations.

`speed`
: Return current calculation speed [iter/sec].

`step`() → None
: Advance the simulation by one step. Returns after the step will have finished.

`stopAtIter`
: Get/set number of iteration after which the simulation will stop.

`subStep`
: Get the current subStep number (only meaningful if O.subStepping==True); -1 when outside the loop, otherwise either 0 (O.subStepping==False) or number of engine to be run (O.subStepping==True)

`subStepping`
: Get/set whether subStepping is active.

`switchScene`() → None
: Switch to alternative simulation (while keeping the old one). Calling the function again switches back to the first one. Note that most variables from the first simulation will still refer to the first simulation even after the switch (e.g. b=O.bodies[4]; O.switchScene(); [b still refers to the body in the first simulation here])

`switchToScene`(*(int)arg2*) → None
: Switch to defined scene. Default scene has number 0, other scenes have to be created by addScene method.

`tags`
: Tags (string=string dictionary) of the current simulation (container supporting string-index access/assignment)

`time`
: Return virtual (model world) time of the simulation.

`timingEnabled`
: Globally enable/disable timing services (see documentation of the timing module).

`tmpFilename`() → str
: Return unique name of file in temporary directory which will be deleted when yade exits.

`tmpToFile`(*(str)fileName*[, *(str)mark=''*]) → None
: Save XML of saveTmp'd simulation into *fileName*.

`tmpToString`([*(str)mark=''*]) → str
: Return XML of saveTmp'd simulation as string.

`trackEnergy`
: When energy tracking is enabled or disabled in this simulation.

`wait`() → None
: Don't return until the simulation will have been paused. (Returns immediately if not running).

7.12.2 BodyContainer

class `yade.wrapper.`**`BodyContainer`**

`__init__`(*(BodyContainer)arg2*) → None

`append`(*(Body)arg2*) → int

Append one Body instance, return its id.

append((BodyContainer)arg1, (object)arg2) → object : Append list of Body instance, return list of ids

`appendClumped`(*(object)arg2*) → tuple
Append given list of bodies as a clump (rigid aggregate); return list of ids.

`clear`() → None
Remove all bodies (interactions not checked)

`clump`(*(object)arg2*) → int
Clump given bodies together (creating a rigid aggregate); returns clump id.

`erase`(*(int)arg2*) → bool
Erase body with the given id; all interaction will be deleted by InteractionLoop in the next step.

`replace`(*(object)arg2*) → object

7.12.3 InteractionContainer

class `yade.wrapper.`**`InteractionContainer`**
Access to interactions of simulation, by using

1.id's of both Bodies of the interactions, e.g. `O.interactions[23,65]`

2.iteraction over the whole container:

```
for i in O.interactions: print i.id1,i.id2
```

Note: Iteration silently skips interactions that are not real.

`__init__`(*(InteractionContainer)arg2*) → None

`clear`() → None
Remove all interactions, and invalidate persistent collider data (if the collider supports it).

`countReal`() → int
Return number of interactions that are "real", i.e. they have phys and geom.

`erase`(*(int)arg2*, *(int)arg3*) → None
Erase one interaction, given by id1, id2 (internally, `requestErase` is called – the interaction might still exist as potential, if the Collider decides so).

`eraseNonReal`() → None
Erase all interactions that are not real .

`nth`(*(int)arg2*) → Interaction
Return n-th interaction from the container (usable for picking random interaction).

`serializeSorted`

`withBody`(*(int)arg2*) → list
Return list of real interactions of given body.

`withBodyAll`(*(int)arg2*) → list
Return list of all (real as well as non-real) interactions of given body.

7.12.4 ForceContainer

class `yade.wrapper.ForceContainer`

`__init__`(*(ForceContainer)arg2*) → None

`addF`(*(int)id*, *(Vector3)f*) → None
Apply force on body (accumulates).

`addMove`(*(int)id*, *(Vector3)m*) → None
Apply displacement on body (accumulates).

`addRot`(*(int)id*, *(Vector3)r*) → None
Apply rotation on body (accumulates).

`addT`(*(int)id*, *(Vector3)t*) → None
Apply torque on body (accumulates).

`f`(*(int)id*) → Vector3
Force applied on body.

`m`(*(int)id*) → Vector3
Deprecated alias for t (torque).

`move`(*(int)id*) → Vector3
Displacement applied on body.

`rot`(*(int)id*) → Vector3
Rotation applied on body.

`syncCount`
Number of synchronizations of ForceContainer (cummulative); if significantly higher than number of steps, there might be unnecessary syncs hurting performance.

`t`(*(int)id*) → Vector3
Torque applied on body.

7.12.5 MaterialContainer

class `yade.wrapper.MaterialContainer`
Container for Materials. A material can be accessed using

1.numerical index in range(0,len(cont)), like cont[2];

2.textual label that was given to the material, like cont['steel']. This etails traversing all materials and should not be used frequently.

`__init__`(*(MaterialContainer)arg2*) → None

`append`(*(Material)arg2*) → int

Add new shared Material; changes its id and return it.

append((MaterialContainer)arg1, (object)arg2) → object : Append list of Material instances, return list of ids.

`index`(*(str)arg2*) → int
Return id of material, given its label.

7.12.6 Scene

class `yade.wrapper.Scene`(*inherits Serializable*)
Object comprising the whole simulation.

`compressionNegative`
Whether the convention is that compression has negative sign (set by Ig2Functor.

`doSort`(*=false*)
Used, when new body is added to the scene.

`dt`(*=1e-8*)
Current timestep for integration.

`flags`(*=0*)
Various flags of the scene; 1 (Scene::LOCAL_COORDS): use local coordinate system rather than global one for per-interaction quantities (set automatically from the functor).

`isPeriodic`(*=false*)
Whether periodic boundary conditions are active.

`iter`(*=0*)
Current iteration (computational step) number

`localCoords`
Whether local coordianate system is used on interactions (set by Ig2Functor.

`selectedBody`(*=-1*)
Id of body that is selected by the user

`speed`(*=0*)
Current calculation speed [iter/s]

`stopAtIter`(*=0*)
Iteration after which to stop the simulation.

`subStep`(*=-1*)
Number of sub-step; not to be changed directly. -1 means to run loop prologue (cell integration), 0…n-1 runs respective engines (n is number of engines), n runs epilogue (increment step number and time.

`subStepping`(*=false*)
Whether we currently advance by one engine in every step (rather than by single run through all engines).

`tags`(*=uninitalized*)
Arbitrary key=value associations (tags like mp3 tags: author, date, version, description etc.)

`time`(*=0*)
Simulation time (virtual time) [s]

`trackEnergy`(*=false*)
Whether energies are being traced.

7.12.7 Cell

class `yade.wrapper.`**`Cell`**(*inherits Serializable*)
Parameters of periodic boundary conditions. Only applies if O.isPeriodic==True.

`hSize`
Base cell vectors (columns of the matrix), updated at every step from velGrad (trsf accumulates applied velGrad transformations). Setting *hSize* during a simulation is not supported by most contact laws, it is only meant to be used at iteration 0 before any interactions have been created.

`hSize0`
Value of untransformed hSize, with respect to current trsf (computed as trsf<Cell.trsf>‘ ^{1} × :yref:‘hSize.

`homoDeformOld`(*=3*)
[DEPRECATED: homoDeform=3 is the only option left, kept here for compatibility] Deform (velGrad) the cell homothetically, by adjusting positions or velocities of particles. The values have the following meaning: 0: no homothetic deformation, 1: set absolute particle positions directly (when `velGrad` is non-zero), but without changing their velocity, 2: adjust particle

velocity (only when `velGrad` changed) with Δv_i=Δ v x_i. 3: as 2, but include a 2nd order term in addition – the derivative of 1 (convective term in the velocity update).

`prevVelGrad`(*=Matrix3r::Zero()*)
Velocity gradient in the previous step.

`refHSize`(*=Matrix3r::Identity()*)
Reference cell configuration, only used with OpenGLRenderer.dispScale. Updated automatically when hSize or trsf is assigned directly; also modified by utils.setRefSe3 (called e.g. by the **:gui:'Reference'** button in the UI).

`refSize`
Reference size of the cell (lengths of initial cell vectors, i.e. column norms of hSize).

Note: Modifying this value is deprecated, use setBox instead.

`setBox`(*(Vector3)arg2*) → None

Set Cell shape to be rectangular, with dimensions along axes specified by given argument. Shorthand for assigning diagonal matrix with respective entries to hSize.

setBox((Cell)arg1, (float)arg2, (float)arg3, (float)arg4) → None : Set Cell shape to be rectangular, with dimensions along x, y, z specified by arguments. Shorthand for assigning diagonal matrix with the respective entries to hSize.

`shearPt`(*(Vector3)arg2*) → Vector3
Apply shear (cell skew+rot) on the point

`shearTrsf`
Current skew+rot transformation (no resize)

`size`
Current size of the cell, i.e. lengths of the 3 cell lateral vectors contained in Cell.hSize columns. Updated automatically at every step.

`trsf`
Current transformation matrix of the cell, obtained from time integration of Cell.velGrad.

`unshearPt`(*(Vector3)arg2*) → Vector3
Apply inverse shear on the point (removes skew+rot of the cell)

`unshearTrsf`
Inverse of the current skew+rot transformation (no resize)

`velGrad`(*=Matrix3r::Zero()*)
Velocity gradient of the transformation; used in NewtonIntegrator. Values of velGrad accumulate in trsf at every step.

`volume`
Current volume of the cell.

`wrap`(*(Vector3)arg2*) → Vector3
Transform an arbitrary point into a point in the reference cell

`wrapPt`(*(Vector3)arg2*) → Vector3
Wrap point inside the reference cell, assuming the cell has no skew+rot.

7.13 Other classes

class `yade.wrapper.Engine`(*inherits Serializable*)
Basic execution unit of simulation, called from the simulation loop (O.engines)

`dead`(*=false*)
If true, this engine will not run at all; can be used for making an engine temporarily deactivated and only resurrect it at a later point.

`execCount`
: Cummulative count this engine was run (only used if O.timingEnabled==`True`).

`execTime`
: Cummulative time this Engine took to run (only used if O.timingEnabled==`True`).

`label`(*=uninitalized*)
: Textual label for this object; must be valid python identifier, you can refer to it directly from python.

`timingDeltas`
: Detailed information about timing inside the Engine itself. Empty unless enabled in the source code and O.timingEnabled==`True`.

class `yade.wrapper.Cell`(*inherits Serializable*)
Parameters of periodic boundary conditions. Only applies if O.isPeriodic==True.

`hSize`
: Base cell vectors (columns of the matrix), updated at every step from velGrad (trsf accumulates applied velGrad transformations). Setting *hSize* during a simulation is not supported by most contact laws, it is only meant to be used at iteration 0 before any interactions have been created.

`hSize0`
: Value of untransformed hSize, with respect to current trsf (computed as trsf<Cell.trsf>‘ ^{1} × :yref:‘hSize.

`homoDeformOld`(*=3*)
: [DEPRECATED: homoDeform=3 is the only option left, kept here for compatibility] Deform (velGrad) the cell homothetically, by adjusting positions or velocities of particles. The values have the following meaning: 0: no homothetic deformation, 1: set absolute particle positions directly (when `velGrad` is non-zero), but without changing their velocity, 2: adjust particle velocity (only when `velGrad` changed) with Δv_i=Δ v x_i. 3: as 2, but include a 2nd order term in addition – the derivative of 1 (convective term in the velocity update).

`prevVelGrad`(*=Matrix3r::Zero()*)
: Velocity gradient in the previous step.

`refHSize`(*=Matrix3r::Identity()*)
: Reference cell configuration, only used with OpenGLRenderer.dispScale. Updated automatically when hSize or trsf is assigned directly; also modified by utils.setRefSe3 (called e.g. by the **:gui:‘Reference‘** button in the UI).

`refSize`
: Reference size of the cell (lengths of initial cell vectors, i.e. column norms of hSize).

Note: Modifying this value is deprecated, use setBox instead.

`setBox`(*(Vector3)arg2*) → None

Set Cell shape to be rectangular, with dimensions along axes specified by given argument. Shorthand for assigning diagonal matrix with respective entries to hSize.

setBox((Cell)arg1, (float)arg2, (float)arg3, (float)arg4) → None : Set Cell shape to be rectangular, with dimensions along x, y, z specified by arguments. Shorthand for assigning diagonal matrix with the respective entries to hSize.

`shearPt`(*(Vector3)arg2*) → Vector3
: Apply shear (cell skew+rot) on the point

`shearTrsf`
: Current skew+rot transformation (no resize)

size
Current size of the cell, i.e. lengths of the 3 cell lateral vectors contained in Cell.hSize columns. Updated automatically at every step.

trsf
Current transformation matrix of the cell, obtained from time integration of Cell.velGrad.

unshearPt(*(Vector3)arg2*) → Vector3
Apply inverse shear on the point (removes skew+rot of the cell)

unshearTrsf
Inverse of the current skew+rot transformation (no resize)

velGrad(*=Matrix3r::Zero()*)
Velocity gradient of the transformation; used in NewtonIntegrator. Values of velGrad accumulate in trsf at every step.

volume
Current volume of the cell.

wrap(*(Vector3)arg2*) → Vector3
Transform an arbitrary point into a point in the reference cell

wrapPt(*(Vector3)arg2*) → Vector3
Wrap point inside the reference cell, assuming the cell has no skew+rot.

class yade.wrapper.**TimingDeltas**

data
Get timing data as list of tuples (label, execTime[nsec], execCount) (one tuple per checkpoint)

reset() → None
Reset timing information

class yade.wrapper.**GlExtraDrawer**(*inherits Serializable*)
Performing arbitrary OpenGL drawing commands; called from OpenGLRenderer (see OpenGLRenderer.extraDrawers) once regular rendering routines will have finished.

This class itself does not render anything, derived classes should override the *render* method.

dead(*=false*)
Deactivate the object (on error/exception).

class yade.wrapper.**GlIGeomDispatcher**(*inherits Dispatcher → Engine → Serializable*)
Dispatcher calling functors based on received argument type(s).

dispFunctor(*(IGeom)arg2*) → GlIGeomFunctor
Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

dispMatrix(*[(bool)names=True]*) → dict
Return dictionary with contents of the dispatch matrix.

functors
Functors associated with this dispatcher.

class yade.wrapper.**ParallelEngine**(*inherits Engine → Serializable*)
Engine for running other Engine in parallel.

__init__() → None
object ___init___(tuple args, dict kwds)

___init___((list)arg2) → object : Construct from (possibly nested) list of slaves.

slaves
List of lists of Engines; each top-level group will be run in parallel with other groups, while Engines inside each group will be run sequentially, in given order.

class yade.wrapper.**GlShapeDispatcher**(*inherits Dispatcher → Engine → Serializable*)
Dispatcher calling functors based on received argument type(s).

dispFunctor(*(Shape)arg2*) → GlShapeFunctor
Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

dispMatrix([*(bool)names=True*]) → dict
Return dictionary with contents of the dispatch matrix.

functors
Functors associated with this dispatcher.

class yade.wrapper.**Functor**(*inherits Serializable*)
Function-like object that is called by Dispatcher, if types of arguments match those the Functor declares to accept.

bases
Ordered list of types (as strings) this functor accepts.

label(*=uninitalized*)
Textual label for this object; must be valid python identifier, you can refer to it directly fron python (must be a valid python identifier).

timingDeltas
Detailed information about timing inside the Dispatcher itself. Empty unless enabled in the source code and O.timingEnabled==True.

class yade.wrapper.**Serializable**

dict() → dict
Return dictionary of attributes.

updateAttrs(*(dict)arg2*) → None
Update object attributes from given dictionary

class yade.wrapper.**GlExtra_LawTester**(*inherits GlExtraDrawer → Serializable*)
Find an instance of LawTester and show visually its data.

tester(*=uninitalized*)
Associated LawTester object.

class yade.wrapper.**GlStateDispatcher**(*inherits Dispatcher → Engine → Serializable*)
Dispatcher calling functors based on received argument type(s).

dispFunctor(*(State)arg2*) → GlStateFunctor
Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

dispMatrix([*(bool)names=True*]) → dict
Return dictionary with contents of the dispatch matrix.

functors
Functors associated with this dispatcher.

class yade.wrapper.**MatchMaker**(*inherits Serializable*)
Class matching pair of ids to return pre-defined (for a pair of ids defined in matches) or derived value (computed using algo) of a scalar parameter. It can be called (`id1`, `id2`, `val1=NaN`, `val2=NaN`) in both python and c++.

Note: There is a *converter* from python number defined for this class, which creates a new MatchMaker returning the value of that number; instead of giving the object instance therefore, you can only pass the number value and it will be converted automatically.

algo
Alogorithm used to compute value when no match for ids is found. Possible values are

- 'avg' (arithmetic average)
- 'min' (minimum value)

- 'max' (maximum value)
- 'harmAvg' (harmonic average)

The following algo algorithms do *not* require meaningful input values in order to work:

- 'val' (return value specified by val)
- 'zero' (always return 0.)

`computeFallback`(*(float)val1*, *(float)val2*) → float
Compute algo value for *val1* and *val2*, using algorithm specified by algo.

`matches`(*=uninitalized*)
Array of `(id1,id2,value)` items; queries matching `id1` + `id2` or `id2` + `id1` will return `value`

`val`(*=NaN*)
Constant value returned if there is no match and algo is `val`

class `yade.wrapper.GlBoundDispatcher`(*inherits Dispatcher → Engine → Serializable*)
Dispatcher calling functors based on received argument type(s).

`dispFunctor`(*(Bound)arg2*) → GlBoundFunctor
Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

`dispMatrix`([*(bool)names=True*]) → dict
Return dictionary with contents of the dispatch matrix.

`functors`
Functors associated with this dispatcher.

class `yade.wrapper.GlIPhysDispatcher`(*inherits Dispatcher → Engine → Serializable*)
Dispatcher calling functors based on received argument type(s).

`dispFunctor`(*(IPhys)arg2*) → GlIPhysFunctor
Return functor that would be dispatched for given argument(s); None if no dispatch; ambiguous dispatch throws.

`dispMatrix`([*(bool)names=True*]) → dict
Return dictionary with contents of the dispatch matrix.

`functors`
Functors associated with this dispatcher.

class `yade.wrapper.GlExtra_OctreeCubes`(*inherits GlExtraDrawer → Serializable*)
Render boxed read from file

`boxesFile`(*=uninitalized*)
File to read boxes from; ascii files with `x0 y0 z0 x1 y1 z1 c` records, where `c` is an integer specifying fill (0 for wire, 1 for filled).

`fillRangeDraw`(*=Vector2i(-2, 2)*)
Range of fill indices that will be rendered.

`fillRangeFill`(*=Vector2i(2, 2)*)
Range of fill indices that will be filled.

`levelRangeDraw`(*=Vector2i(-2, 2)*)
Range of levels that will be rendered.

`noFillZero`(*=true*)
Do not fill 0-fill boxed (those that are further subdivided)

class `yade.wrapper.Dispatcher`(*inherits Engine → Serializable*)
Engine dispatching control to its associated functors, based on types of argument it receives. This abstract base class provides no functionality in itself.

class `yade.wrapper.`**`EnergyTracker`**(*inherits Serializable*)

Storage for tracing energies. Only to be used if O.traceEnergy is True.

`clear``()` → None

Clear all stored values.

`energies`(*=uninitalized*)

Energy values, in linear array

`items``()` → list

Return contents as list of (name,value) tuples.

`keys``()` → list

Return defined energies.

`total``()` → float

Return sum of all energies.

Chapter 8

Yade modules

8.1 yade.bodiesHandling module

Miscellaneous functions, which are useful for handling bodies.

`yade.bodiesHandling.sphereDuplicate`(*idSphere*)

The functions makes a copy of sphere

`yade.bodiesHandling.spheresModify`(*idSpheres=[ ], mask=-1, shift=Vector3(0, 0, 0), scale=1.0, orientation=Quaternion((1, 0, 0), 0), copy=False*)

The function accepts the list of spheres id's or list of bodies and modifies them: rotating, scaling, shifting. if copy=True copies bodies and modifies them. Also the mask can be given. If idSpheres not empty, the function affects only bodies, where the mask passes. If idSpheres is empty, the function search for bodies, where the mask passes.

Parameters

***shift*: Vector3** Vector3(X,Y,Z) parameter moves spheres.

***scale*: float** factor scales given spheres.

***orientation*: quaternion** orientation of spheres

***mask*: int** Body.mask for the checked bodies

Returns list of bodies if copy=True, and Boolean value if copy=False

`yade.bodiesHandling.spheresPackDimensions`(*idSpheres=[ ], mask=-1*)

The function accepts the list of spheres id's or list of bodies and calculates max and min dimensions, geometrical center.

Parameters

- **idSpheres** (*list*) – list of spheres
- **mask** (*int*) – Body.mask for the checked bodies

Returns dictionary with keys `min` (minimal dimension, Vector3), `max` (maximal dimension, Vector3), `minId` (minimal dimension sphere Id, Vector3), `maxId` (maximal dimension sphere Id, Vector3), `center` (central point of bounding box, Vector3), `extends` (sizes of bounding box, Vector3), `volume` (volume of spheres, Real), `mass` (mass of spheres, Real), `number` (number of spheres, int),

8.2 yade.eudoxos module

Miscillaneous functions that are not believed to be generally usable, therefore kept in my "private" module here.

They comprise notably oofem export and various CPM-related functions.

class `yade.eudoxos.`**`IntrSmooth3d`**

Return spatially weigted gaussian average of arbitrary quantity defined on interactions.

At construction time, all real interactions are put inside spatial grid, permitting fast search for points in neighbourhood defined by distance.

Parameters for the distribution are standard deviation σ and relative cutoff distance *relThreshold* (3 by default) which will discard points farther than *relThreshold* $\times\sigma$.

Given central point p_0, points are weighted by gaussian function

$$\rho(p_0, p) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(\frac{-||p_0 - p||^2}{2\sigma^2}\right)$$

To get the averaged value, simply call the instance, passing central point and callable object which received interaction object and returns the desired quantity:

```
>>> O.reset()
>>> from yade import utils
>>> O.bodies.append([utils.sphere((0,0,0),1),utils.sphere((0,0,1.9),1)])
[0, 1]
>>> O.engines=[InteractionLoop([Ig2_Sphere_Sphere_Dem3DofGeom(),],[Ip2_FrictMat_FrictMat_FrictPhys()]
>>> utils.createInteraction(0,1)
<Interaction instance at 0x...>
```

>> is3d=IntrSmooth3d(0.003) >> is3d((0,0,0),lambda i: i.phys.normalForce) Vector3(0,0,0)

`bounds``()`

`count``()`

`yade.eudoxos.`**`displacementsInteractionsExport`**`(`*fName*`)`

`yade.eudoxos.`**`eliminateJumps`**`(`*eps*, *sigma*, *numSteep=10*, *gapWidth=5*, *movWd=40*`)`

`yade.eudoxos.`**`estimatePoissonYoung`**`(`*principalAxis*, *stress=0*, *plot=False*, *cutoff=0.0*`)`

Estimate Poisson's ration given the "principal" axis of straining. For every base direction, homogenized strain is computed (slope in linear regression on discrete function particle coordinate → → particle displacement in the same direction as returned by utils.coordsAndDisplacements) and, (if axis '0' is the strained axis) the poisson's ratio is given as -½(ε1+ε2)/ε .

Young's modulus is computed as σ/ε ; if stress σ is not given (default 0), the result is 0.

cutoff, if > 0., will take only smaller part (centered) or the specimen into account

`yade.eudoxos.`**`estimateStress`**`(`*strain*, *cutoff=0.0*`)`

Use summed stored energy in contacts to compute macroscopic stress over the same volume, provided known strain.

`yade.eudoxos.`**`oofemDirectExport`**`(`*fileBase*, *title=None*, *negIds=[]*, *posIds=[]*`)`

`yade.eudoxos.`**`oofemPrescribedDisplacementsExport`**`(`*fileName*`)`

`yade.eudoxos.`**`oofemTextExport`**`(`*fName*`)`

Export simulation data in text format

The format is line-oriented as follows:

```
E G                                                 # elastic material parameters
epsCrackOnset crackOpening xiShear transStrainCoeff # tensile parameters; epsFr=crackOpening/len
cohesionT tanPhi                                    # shear parameters
number_of_spheres number_of_links
```

```
id x y z r boundary                                  # spheres; boundary: -1 negative, 0 none, 1 positive
…
id1 id2 cp_x cp_y cp_z A                             # interactions; cp = contact point; A = cross-section
```

class `yade._eudoxos.`**`HelixInteractionLocator2d`**

Locate all real interactions in 2d plane (reduced by spiral projection from 3d, using `Shop::spiralProject`, which is the same as utils.spiralProject) using their contact points.

Note: Do not run simulation while using this object.

`__init__`(*(float)dH_dTheta*[, *(int)axis=0*[, *(float)periodStart=nan*[, *(float)theta0=0*[, *(float)thetaMin=nan*[, *(float)thetaMax=nan*]]]]]) → None

Parameters

- **dH_dTheta** (*float*) – Spiral inclination, i.e. height increase per 1 radian turn;
- **axis** (*int*) – axis of rotation (0=x,1=y,2=z)
- **theta** (*float*) – spiral angle at zero height (theta intercept)
- **thetaMin** (*float*) – only interactions with ϑ *thetaMin* will be considered (NaN to deactivate)
- **thetaMax** (*float*) – only interactions with ϑ *thetaMax* will be considered (NaN to deactivate)

See utils.spiralProject.

`hi`

Return upper corner of the rectangle containing all interactions.

`intrsAroundPt`(*(Vector2)pt2d*, *(float)radius*) → list

Return list of interaction objects that are not further from *pt2d* than *radius* in the projection plane

`lo`

Return lower corner of the rectangle containing all interactions.

`macroAroundPt`(*(Vector2)pt2d*, *(float)radius*) → tuple

Compute macroscopic stress around given point; the interaction (n and σ^T are rotated to the projection plane by ϑ (as given by utils.spiralProject) first, but no skew is applied). The formula used is

$$\sigma_{ij} = \frac{1}{V}\sum_{IJ} d^{IJ}A^{IJ}\left[\sigma^{N,IJ}n_i^{IJ}n_j^{IJ} + \frac{1}{2}\left(\sigma_i^{T,IJ}n_j^{IJ} + \sigma_j^{T,IJ}n_i^{IJ}\right)\right]$$

where the sum is taken over volume V containing interactions IJ between spheres I and J;

- i, j indices denote Cartesian components of vectors and tensors,
- d^{IJ} is current distance between spheres I and J,
- A^{IJ} is area of contact IJ,
- n is (ϑ-rotated) interaction normal (unit vector pointing from center of I to the center of J)
- $\sigma^{N,IJ}$ is normal stress (as scalar) in contact IJ,
- $\sigma^{T,IJ}$ is shear stress in contact IJ in global coordinates and ϑ-rotated.

Additionally, computes average of CpmPhys.omega ($\bar{\omega}$) and CpmPhys.kappaD ($\bar{\kappa}_D$). *N* is the number of interactions in the volume given.

Returns tuple of (N, $\boldsymbol{\sigma}$, $\bar{\omega}$, $\bar{\kappa}_D$).

class `yade._eudoxos.`**`InteractionLocator`**

Locate all (real) interactions in space by their contact point. When constructed, all real interactions are spatially indexed (uses vtkPointLocator internally). Use instance methods to use those data.

Note: Data might become inconsistent with real simulation state if simulation is being run between creation of this object and spatial queries.

`bounds`

Return coordinates of lower and uppoer corner of axis-aligned abounding box of all interactions

`count`

Number of interactions held

`intrsAroundPt`(*(Vector3)point*, *(float)maxDist*) → list

Return list of real interactions that are not further than *maxDist* from *point*.

`macroAroundPt`(*(Vector3)point*, *(float)maxDist*[, *(float)forceVolume=-1*]) → tuple

Return tuple of averaged stress tensor (as Matrix3), average omega and average kappa values. *forceVolume* can be used (if positive) rather than the sphere (with *maxDist* radius) volume for the computation. (This is useful if *point* and *maxDist* encompass empty space that you want to avoid.)

`yade._eudoxos.`**`particleConfinement`**() → None

`yade._eudoxos.`**`velocityTowardsAxis`**(*(Vector3)axisPoint*, *(Vector3)axisDirection*, *(float)timeToAxis*[, *(float)subtractDist*[, *(float)perturbation*]]) → None

8.3 yade.export module

Export (not only) geometry to various formats.

class `yade.export.`**`VTKExporter`**

Class for exporting data to VTK Simple Legacy File (for example if, for some reason, you are not able to use VTKRecorder). Export of spheres and facets is supported.

USAGE: create object vtkExporter = VTKExporter('baseFileName'), add to engines PyRunner with command='vtkWriter.exportSomething(params)'

`exportFacets`()

exports facets (positions) and defined properties.

:parameters: 'ids': list | "all" if "all", then export all spheres, otherwise only spheres from integer list 'what': [tuple(2)] see exportSpheres 'comment': string comment to add to vtk file 'numLabel': int number of file (e.g. time step), if unspecified, the last used value + 1 will be used

`exportSpheres`()

exports spheres (positions and radius) and defined properties.

:parameters: 'ids': [int] | "all" if "all", then export all spheres, otherwise only spheres from integer list 'what': [tuple(2)] what other than then position and radius export. parameter is list of couple (name,command). Name is string under which it is save to vtk, command is string to evaluate. Node that the bodies are labeled as b in this function. Scalar, vector and tensor variables are supported. For example, to export velocity (with name particleVelocity) and the distance form point (0,0,0) (named as dist) you should write: ... what=[('particleVelocity','b.state.vel'),('dist','b.state.pos.norm()', ... 'comment': string comment to add to vtk file 'numLabel': int number of file (e.g. time step), if unspecified, the last used value + 1 will be used

class `yade.export.`**`VTKWriter`**

USAGE: create object vtk_writer = VTKWriter('base_file_name'), add to engines PyRunner with command='vtk_writer.snapshot()'

`snapshot()`

`yade.export.text`(*filename*, *mask=-1*)

Save sphere coordinates into a text file; the format of the line is: x y z r. Non-spherical bodies are silently skipped. Example added to examples/regular-sphere-pack/regular-sphere-pack.py :parameters: *filename*: string

the name of the file, where sphere coordinates will be exported.

***mask*:** export only spheres with the corresponding mask

Returns number of spheres which were written.

`yade.export.textExt`(*filename*, *format='x_y_z_r'*, *comment=''*, *mask=-1*)

Save sphere coordinates and other parameters into a text file in specific format. Non-spherical bodies are silently skipped. Users can add here their own specific format, giving meaningful names. The first file row will contain the format name. Be sure to add the same format specification in ymport.textExt.

param string filename the name of the file, where sphere coordinates will be exported.

param string format the name of output format. Supported *x_y_z_r'(default), 'x_y_z_r_matId*

param string comment the text, which will be added as a comment at the top of file. If you want to create several lines of text, please use '

#' for next lines.

param int mask' export only spheres with the corresponding mask export only spheres with the corresponding mask

Return number of spheres which were written.

8.4 yade.geom module

Creates geometry objects from facets.

`yade.geom.facetBox`(*center*, *extents*, *orientation=Quaternion((1, 0, 0), 0)*, *wallMask=63*, ***kw*)

Create arbitrarily-aligned box composed of facets, with given center, extents and orientation. If any of the box dimensions is zero, corresponding facets will not be created. The facets are oriented outwards from the box.

Parameters

- **center** (*Vector3*) – center of the box
- **extents** (*Vector3*) – lengths of the box sides
- **orientation** (*Quaternion*) – orientation of the box
- **wallMask** (*bitmask*) – determines which walls will be created, in the order -x (1), +x (2), -y (4), +y (8), -z (16), +z (32). The numbers are ANDed; the default 63 means to create all walls
- ****kw** – (unused keyword arguments) passed to utils.facet

Returns list of facets forming the box

`yade.geom.facetBunker`(*center*, *dBunker*, *dOutput*, *hBunker*, *hOutput*, *hPipe=0.0*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *wallMask=4*, *angleRange=None*, *closeGap=False*, ***kw*)

Create arbitrarily-aligned bunker, composed of facets, with given center, radii, heights and orientation. Return List of facets forming the bunker;

```
   dBunker

______________
|            |
|            |
|            | hBunker
|            |
|            |
|            |
|____________|
\            /
 \          /
  \        /   hOutput
   \      /
    \____/
    |    |
    |____|     hPipe
    dOutput
```

Parameters

- **center** (*Vector3*) – center of the created bunker
- **dBunker** (*float*) – bunker diameter, top
- **dOutput** (*float*) – bunker output diameter
- **hBunker** (*float*) – bunker height
- **hOutput** (*float*) – bunker output height
- **hPipe** (*float*) – bunker pipe height
- **orientation** (*Quaternion*) – orientation of the bunker; the reference orientation has axis along the +x axis.
- **segmentsNumber** (*int*) – number of edges on the bunker surface (>=5)
- **wallMask** (*bitmask*) – determines which walls will be created, in the order up (1), down (2), side (4). The numbers are ANDed; the default 7 means to create all walls
- **angleRange** (*(ϑmin,Θmax)*) – allows one to create only part of bunker by specifying range of angles; if `None`, (0,2*pi) is assumed.
- **closeGap** (*bool*) – close range skipped in angleRange with triangular facets at cylinder bases.
- ****kw** – (unused keyword arguments) passed to utils.facet;

`yade.geom.`**`facetCone`**(*center*, *radiusTop*, *radiusBottom*, *height*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *wallMask=7*, *angleRange=None*, *closeGap=False*, ***kw*)

Create arbitrarily-aligned cone composed of facets, with given center, radius, height and orientation. Return List of facets forming the cone;

Parameters

- **center** (*Vector3*) – center of the created cylinder
- **radiusTop** (*float*) – cone top radius
- **radiusBottom** (*float*) – cone bottom radius
- **height** (*float*) – cone height
- **orientation** (*Quaternion*) – orientation of the cone; the reference orientation has axis along the +x axis.
- **segmentsNumber** (*int*) – number of edges on the cone surface (>=5)

- **wallMask** (*bitmask*) – determines which walls will be created, in the order up (1), down (2), side (4). The numbers are ANDed; the default 7 means to create all walls
- **angleRange** (*(ϑmin,Θmax)*) – allows one to create only part of cone by specifying range of angles; if `None`, (0,2*pi) is assumed.
- **closeGap** (*bool*) – close range skipped in angleRange with triangular facets at cylinder bases.
- ****kw** – (unused keyword arguments) passed to utils.facet;

`yade.geom.`**`facetCylinder`**(*center*, *radius*, *height*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *wallMask=7*, *angleRange=None*, *closeGap=False*, ***kw*)

Create arbitrarily-aligned cylinder composed of facets, with given center, radius, height and orientation. Return List of facets forming the cylinder;

Parameters

- **center** (*Vector3*) – center of the created cylinder
- **radius** (*float*) – cylinder radius
- **height** (*float*) – cylinder height
- **orientation** (*Quaternion*) – orientation of the cylinder; the reference orientation has axis along the +x axis.
- **segmentsNumber** (*int*) – number of edges on the cylinder surface (>=5)
- **wallMask** (*bitmask*) – determines which walls will be created, in the order up (1), down (2), side (4). The numbers are ANDed; the default 7 means to create all walls
- **angleRange** (*(ϑmin,Θmax)*) – allows one to create only part of bunker by specifying range of angles; if `None`, (0,2*pi) is assumed.
- **closeGap** (*bool*) – close range skipped in angleRange with triangular facets at cylinder bases.
- ****kw** – (unused keyword arguments) passed to utils.facet;

`yade.geom.`**`facetCylinderConeGenerator`**(*center*, *radiusTop*, *height*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *wallMask=7*, *angleRange=None*, *closeGap=False*, *radiusBottom=-1*, ***kw*)

Please, do not use this function directly! Use geom.facetCylinder and geom.facetCone instead. This is the base function for generating cylinders and cones from facets. :param float radiusTop: top radius :param float radiusBottom: bottom radius :param **kw: (unused keyword arguments) passed to utils.facet;

`yade.geom.`**`facetHelix`**(*center*, *radiusOuter*, *pitch*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *angleRange=None*, *radiusInner=0*, ***kw*)

Create arbitrarily-aligned helix composed of facets, with given center, radius (outer and inner), pitch and orientation. Return List of facets forming the helix;

Parameters

- **center** (*Vector3*) – center of the created cylinder
- **radiusOuter** (*float*) – outer radius
- **radiusInner** (*float*) – inner height (can be 0)
- **orientation** (*Quaternion*) – orientation of the helix; the reference orientation has axis along the +x axis.
- **segmentsNumber** (*int*) – number of edges on the helix surface (>=3)
- **angleRange** (*(ϑmin,Θmax)*) – range of angles; if `None`, (0,2*pi) is assumed.
- ****kw** – (unused keyword arguments) passed to utils.facet;

`yade.geom.facetParallelepiped`(*center*, *extents*, *height*, *orientation=Quaternion((1, 0, 0), 0)*, *wallMask=63*, ***kw*)

Create arbitrarily-aligned Parallelepiped composed of facets, with given center, extents, height and orientation. If any of the parallelepiped dimensions is zero, corresponding facets will not be created. The facets are oriented outwards from the parallelepiped.

Parameters

- **center** (*Vector3*) – center of the parallelepiped
- **extents** (*Vector3*) – lengths of the parallelepiped sides
- **height** (*Real*) – height of the parallelepiped (along axis z)
- **orientation** (*Quaternion*) – orientation of the parallelepiped
- **wallMask** (*bitmask*) – determines which walls will be created, in the order -x (1), +x (2), -y (4), +y (8), -z (16), +z (32). The numbers are ANDed; the default 63 means to create all walls
- ****kw** – (unused keyword arguments) passed to utils.facet

Returns list of facets forming the parallelepiped

`yade.geom.facetPolygon`(*center*, *radiusOuter*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *angleRange=None*, *radiusInner=0*, ***kw*)

Create arbitrarily-aligned polygon composed of facets, with given center, radius (outer and inner) and orientation. Return List of facets forming the polygon;

Parameters

- **center** (*Vector3*) – center of the created cylinder
- **radiusOuter** (*float*) – outer radius
- **radiusInner** (*float*) – inner height (can be 0)
- **orientation** (*Quaternion*) – orientation of the polygon; the reference orientation has axis along the +x axis.
- **segmentsNumber** (*int*) – number of edges on the polygon surface (>=3)
- **angleRange** (*(ϑmin,Θmax)*) – allows one to create only part of polygon by specifying range of angles; if `None`, (0,2*pi) is assumed.
- ****kw** – (unused keyword arguments) passed to utils.facet;

`yade.geom.facetPolygonHelixGenerator`(*center*, *radiusOuter*, *pitch=0*, *orientation=Quaternion((1, 0, 0), 0)*, *segmentsNumber=10*, *angleRange=None*, *radiusInner=0*, ***kw*)

Please, do not use this function directly! Use geom.facetPloygon and geom.facetHelix instead. This is the base function for generating polygons and helixes from facets.

8.5 yade.linterpolation module

Module for rudimentary support of manipulation with piecewise-linear functions (which are usually interpolations of higher-order functions, whence the module name). Interpolation is always given as two lists of the same length, where the x-list must be increasing.

Periodicity is supported by supposing that the interpolation can wrap from the last x-value to the first x-value (which should be 0 for meaningful results).

Non-periodic interpolation can be converted to periodic one by padding the interpolation with constant head and tail using the sanitizeInterpolation function.

There is a c++ template function for interpolating on such sequences in pkg/common/Engine/PartialEngine/LinearInterpolate.hpp (stateful, therefore fast for sequential reads).

TODO: Interpolating from within python is not (yet) supported.

`yade.linterpolation.integral`(*x*, *y*)
: Return integral of piecewise-linear function given by points x0,x1,… and y0,y1,…

`yade.linterpolation.revIntegrateLinear`(*I*, *x0*, *y0*, *x1*, *y1*)
: Helper function, returns value of integral variable x for linear function f passing through (x0,y0),(x1,y1) such that 1. x [x0,x1] 2. _x0^x f dx=I and raise exception if such number doesn't exist or the solution is not unique (possible?)

`yade.linterpolation.sanitizeInterpolation`(*x*, *y*, *x0*, *x1*)
: Extends piecewise-linear function in such way that it spans at least the x0…x1 interval, by adding constant padding at the beginning (using y0) and/or at the end (using y1) or not at all.

`yade.linterpolation.xFractionalFromIntegral`(*integral*, *x*, *y*)
: Return x within range x0…xn such that _x0^x f dx==integral. Raises error if the integral value is not reached within the x-range.

`yade.linterpolation.xFromIntegral`(*integralValue*, *x*, *y*)
: Return x such that _x0^x f dx==integral. x wraps around at xn. For meaningful results, therefore, x0 should == 0

8.6 yade.log module

8.7 yade.pack module

Creating packings and filling volumes defined by boundary representation or constructive solid geometry.

For examples, see

- scripts/test/gts-operators.py
- scripts/test/gts-random-pack-obb.py
- scripts/test/gts-random-pack.py
- scripts/test/pack-cloud.py
- scripts/test/pack-predicates.py
- examples/packs/packs.py
- examples/gts-horse/gts-horse.py
- examples/WireMatPM/wirepackings.py

`yade.pack.SpherePack_toSimulation`(*self*, *rot=Matrix3(1, 0, 0, 0, 1, 0, 0, 0, 1)*, ***kw*)
: Append spheres directly to the simulation. In addition calling O.bodies.append, this method also appropriately sets periodic cell information of the simulation.

```
>>> from yade import pack; from math import *
>>> sp=pack.SpherePack()
```

Create random periodic packing with 20 spheres:

```
>>> sp.makeCloud((0,0,0),(5,5,5),rMean=.5,rRelFuzz=.5,periodic=True,num=20)
20
```

Virgin simulation is aperiodic:

```
>>> O.reset()
>>> O.periodic
False
```

Add generated packing to the simulation, rotated by 45° along +z

```
>>> sp.toSimulation(rot=Quaternion((0,0,1),pi/4),color=(0,0,1))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Periodic properties are transferred to the simulation correctly, including rotation (this could be avoided by explicitly passing "hSize=O.cell.hSize" as an argument):

```
>>> O.periodic
True
>>> O.cell.refSize
Vector3(5,5,5)
>>> O.cell.hSize
Matrix3(3.53553,-3.53553,0, 3.53553,3.53553,0, 0,0,5)
```

The current state (even if rotated) is taken as mechanically undeformed, i.e. with identity transformation:

```
>>> O.cell.trsf
Matrix3(1,0,0, 0,1,0, 0,0,1)
```

Parameters

- **rot** (*Quaternion/Matrix3*) – rotation of the packing, which will be applied on spheres and will be used to set Cell.trsf as well.
- ****kw** – passed to utils.sphere

Returns list of body ids added (like O.bodies.append)

`yade.pack.filterSpherePack`(*predicate*, *spherePack*, *returnSpherePack=None*, ***kw*)
Using given SpherePack instance, return spheres the satisfy predicate. The packing will be recentered to match the predicate and warning is given if the predicate is larger than the packing.

`yade.pack.gtsSurface2Facets`(*surf*, ***kw*)
Construct facets from given GTS surface. **kw is passed to utils.facet.

`yade.pack.gtsSurfaceBestFitOBB`(*surf*)
Return (Vector3 center, Vector3 halfSize, Quaternion orientation) describing best-fit oriented bounding box (OBB) for the given surface. See cloudBestFitOBB for details.

`yade.pack.hexaNet`(*radius*, *cornerCoord=[0, 0, 0]*, *xLength=1.0*, *yLength=0.5*, *mos=0.08*, *a=0.04*, *b=0.04*, *startAtCorner=True*, *isSymmetric=False*, ***kw*)
Definition of the particles for a hexagonal wire net in the x-y-plane for the WireMatPM.

Parameters

- **radius** – radius of the particle
- **cornerCoord** – coordinates of the lower left corner of the net
- **xLenght** – net length in x-direction
- **yLenght** – net length in y-direction
- **mos** – mesh opening size (horizontal distance between the double twists)
- **a** – length of double-twist
- **b** – height of single wire section
- **startAtCorner** – if true the generation starts with a double-twist at the lower left corner
- **isSymmetric** – defines if the net is symmetric with respect to the y-axis

Returns set of spheres which defines the net (net) and exact dimensions of the net (lx,ly).

note:: This packing works for the WireMatPM only. The particles at the corner are always generated first. For examples on how to use this packing see examples/WireMatPM. In order to create the proper interactions for the net the interaction radius has to be adapted in the simulation.

class `yade.pack.inGtsSurface_py`(*inherits Predicate*)
This class was re-implemented in c++, but should stay here to serve as reference for implementing Predicates in pure python code. C++ allows us to play dirty tricks in GTS which are not accessible

through pygts itself; the performance penalty of pygts comes from fact that if constructs and destructs bb tree for the surface at every invocation of gts.Point().is_inside(). That is cached in the c++ code, provided that the surface is not manipulated with during lifetime of the object (user's responsibility).

—

Predicate for GTS surfaces. Constructed using an already existing surfaces, which must be closed.

import gts surf=gts.read(open('horse.gts')) inGtsSurface(surf)

Note: Padding is optionally supported by testing 6 points along the axes in the pad distance. This must be enabled in the ctor by saying doSlowPad=True. If it is not enabled and pad is not zero, warning is issued.

`aabb()`

class `yade.pack.inSpace(`*inherits Predicate*`)`

Predicate returning True for any points, with infinite bounding box.

`aabb()`

`center()`

`dim()`

`yade.pack.randomDensePack(`*predicate*, *radius*, *material=-1*, *dim=None*, *cropLayers=0*, *rRelFuzz=0.0*, *spheresInCell=0*, *memoizeDb=None*, *useOBB=True*, *memoDbg=False*, *color=None*, *returnSpherePack=None*`)`

Generator of random dense packing with given geometry properties, using TriaxialTest (aperiodic) or PeriIsoCompressor (periodic). The periodicity depens on whether the spheresInCell parameter is given.

O.switchScene() magic is used to have clean simulation for TriaxialTest without deleting the original simulation. This function therefore should never run in parallel with some code accessing your simulation.

Parameters

- **predicate** – solid-defining predicate for which we generate packing
- **spheresInCell** – if given, the packing will be periodic, with given number of spheres in the periodic cell.
- **radius** – mean radius of spheres
- **rRelFuzz** – relative fuzz of the radius – e.g. radius=10, rRelFuzz=.2, then spheres will have radii 10 ± (10*.2)). 0 by default, meaning all spheres will have exactly the same radius.
- **cropLayers** – (aperiodic only) how many layers of spheres will be added to the computed dimension of the box so that there no (or not so much, at least) boundary effects at the boundaries of the predicate.
- **dim** – dimension of the packing, to override dimensions of the predicate (if it is infinite, for instance)
- **memoizeDb** – name of sqlite database (existent or nonexistent) to find an already generated packing or to store the packing that will be generated, if not found (the technique of caching results of expensive computations is known as memoization). Fuzzy matching is used to select suitable candidate – packing will be scaled, rRelFuzz and dimensions compared. Packing that are too small are dictarded. From the remaining candidate, the one with the least number spheres will be loaded and returned.
- **useOBB** – effective only if a inGtsSurface predicate is given. If true (default), oriented bounding box will be computed first; it can reduce substantially num-

ber of spheres for the triaxial compression (like 10× depending on how much asymmetric the body is), see scripts/test/gts-triax-pack-obb.py.

- **memoDbg** – show packigns that are considered and reasons why they are rejected/accepted
- **returnSpherePack** – see filterSpherePack

Returns SpherePack object with spheres, filtered by the predicate.

`yade.pack.`**`randomPeriPack`**`(`*radius*, *initSize*, *rRelFuzz=0.0*, *memoizeDb=None*, *noPrint=False*`)`

Generate periodic dense packing.

A cell of initSize is stuffed with as many spheres as possible, then we run periodic compression with PeriIsoCompressor, just like with randomDensePack.

Parameters

- **radius** – mean sphere radius
- **rRelFuzz** – relative fuzz of sphere radius (equal distribution); see the same param for randomDensePack.
- **initSize** – initial size of the periodic cell.

Returns SpherePack object, which also contains periodicity information.

`yade.pack.`**`regularHexa`**`(`*predicate*, *radius*, *gap*, ***kw*`)`

Return set of spheres in regular hexagonal grid, clipped inside solid given by predicate. Created spheres will have given radius and will be separated by gap space.

`yade.pack.`**`regularOrtho`**`(`*predicate*, *radius*, *gap*, ***kw*`)`

Return set of spheres in regular orthogonal grid, clipped inside solid given by predicate. Created spheres will have given radius and will be separated by gap space.

`yade.pack.`**`revolutionSurfaceMeridians`**`(`*sects*, *angles*, *origin=Vector3(0, 0, 0)*, *orientation=Quaternion((1, 0, 0), 0)*`)`

Revolution surface given sequences of 2d points and sequence of corresponding angles, returning sequences of 3d points representing meridian sections of the revolution surface. The 2d sections are turned around z-axis, but they can be transformed using the origin and orientation arguments to give arbitrary orientation.

`yade.pack.`**`sweptPolylines2gtsSurface`**`(`*pts*, *threshold=0*, *capStart=False*, *capEnd=False*`)`

Create swept suface (as GTS triangulation) given same-length sequences of points (as 3-tuples).

If threshold is given (>0), then

- degenerate faces (with edges shorter than threshold) will not be created
- gts.Surface().cleanup(threshold) will be called before returning, which merges vertices mutually closer than threshold. In case your pts are closed (last point concident with the first one) this will the surface strip of triangles. If you additionally have capStart==True and capEnd==True, the surface will be closed.

Note: capStart and capEnd make the most naive polygon triangulation (diagonals) and will perhaps fail for non-convex sections.

Warning: the algorithm connects points sequentially; if two polylines are mutually rotated or have inverse sense, the algorithm will not detect it and connect them regardless in their given order.

Creation, manipulation, IO for generic sphere packings.

class `yade._packSpheres.`**`SpherePack`**

Set of spheres represented as centers and radii. This class is returned by pack.randomDensePack, pack.randomPeriPack and others. The object supports iteration over spheres, as in

```
>>> sp=SpherePack()
>>> for center,radius in sp: print center,radius

>>> for sphere in sp: print sphere[0],sphere[1]   ## same, but without unpacking the tuple automatically

>>> for i in range(0,len(sp)): print sp[i][0], sp[i][1]   ## same, but accessing spheres by index
```

Special constructors

Construct from list of `[(c1,r1),(c2,r2),…]`. To convert two same-length lists of `centers` and `radii`, construct with `zip(centers,radii)`.

`__init__`(*[(list)list]*) → None
: Empty constructor, optionally taking list [((cx,cy,cz),r), …] for initial data.

`aabb`() → tuple
: Get axis-aligned bounding box coordinates, as 2 3-tuples.

`add`(*(Vector3)arg2*, *(float)arg3*) → None
: Add single sphere to packing, given center as 3-tuple and radius

`appliedPsdScaling`
: A factor between 0 and 1, uniformly applied on all sizes of of the PSD.

`cellFill`(*(Vector3)arg2*) → None
: Repeat the packing (if periodic) so that the results has dim() >= given size. The packing retains periodicity, but changes cellSize. Raises exception for non-periodic packing.

`cellRepeat`(*(Vector3i)arg2*) → None
: Repeat the packing given number of times in each dimension. Periodicity is retained, cellSize changes. Raises exception for non-periodic packing.

`cellSize`
: Size of periodic cell; is Vector3(0,0,0) if not periodic. (Change this property only if you know what you're doing).

`center`() → Vector3
: Return coordinates of the bounding box center.

`dim`() → Vector3
: Return dimensions of the packing in terms of aabb(), as a 3-tuple.

`fromList`(*(list)arg2*) → None

  Make packing from given list, same format as for constructor. Discards current data.

  fromList((SpherePack)arg1, (object)centers, (object)radii) → None : Make packing from given list, same format as for constructor. Discards current data.

`fromSimulation`() → None
: Make packing corresponding to the current simulation. Discards current data.

`getClumps`() → tuple
: Return lists of sphere ids sorted by clumps they belong to. The return value is (standalones,[clump1,clump2,…]), where each item is list of id's of spheres.

`hasClumps`() → bool
: Whether this object contains clumps.

`isPeriodic`
: was the packing generated in periodic boundaries?

`load`(*(str)fileName*) → None
: Load packing from external text file (current data will be discarded).

makeCloud([*(Vector3)minCorner=Vector3(0, 0, 0)*[, *(Vector3)maxCorner=Vector3(0, 0, 0)*[, *(float)rMean=-1*[, *(float)rRelFuzz=0*[, *(int)num=-1*[, *(bool)periodic=False*[, *(float)porosity=0.65*[, *(object)psdSizes=[]*[, *(object)psdCumm=[]*[, *(bool)distributeMass=False*[, *(int)seed=0*[, *(Matrix3)hSize=Matrix3(0, 0, 0, 0, 0, 0, 0, 0, 0)*]]]]]]]]]]]]) → int

Create random loose packing enclosed in a parallelepiped (also works in 2D if minCorner[k]=maxCorner[k] for one coordinate). Sphere radius distribution can be specified using one of the following ways:

1. *rMean*, *rRelFuzz* and *num* gives uniform radius distribution in *rMean* $(1 \pm$ *rRelFuzz* $)$. Less than *num* spheres can be generated if it is too high.
2. *rRelFuzz*, *num* and (optional) *porosity*, which estimates mean radius so that *porosity* is attained at the end. *rMean* must be less than 0 (default). *porosity* is only an initial guess for the generation algorithm, which will retry with higher porosity until the prescibed *num* is obtained.
3. *psdSizes* and *psdCumm*, two arrays specifying points of the particle size distribution function. As many spheres as possible are generated.
4. *psdSizes*, *psdCumm*, *num*, and (optional) *porosity*, like above but if *num* is not obtained, *psdSizes* will be scaled down uniformly, until *num* is obtained (see appliedPsdScaling).

By default (with `distributeMass==False`), the distribution is applied to particle radii. The usual sense of "particle size distribution" is the distribution of *mass fraction* (rather than particle count); this can be achieved with `distributeMass=True`.

If *num* is defined, then sizes generation is deterministic, giving the best fit of target distribution. It enables spheres placement in descending size order, thus giving lower porosity than the random generation.

Parameters

- **minCorner** (*Vector3*) – lower corner of an axis-aligned box
- **maxCorner** (*Vector3*) – upper corner of an axis-aligned box
- **hSize** (*Matrix3*) – base vectors of a generalized box (arbitrary parallelepiped, typically Cell::hSize), superseeds minCorner and maxCorner if defined. For periodic boundaries only.
- **rMean** (*float*) – mean radius or spheres
- **rRelFuzz** (*float*) – dispersion of radius relative to rMean
- **num** (*int*) – number of spheres to be generated. If negavite (default), generate as many as possible with stochastic sizes, ending after a fixed number of tries to place the sphere in space, else generate exactly *num* spheres with deterministic size distribution.
- **periodic** (*bool*) – whether the packing to be generated should be periodic
- **porosity** (*float*) – initial guess for the iterative generation procedure (if *num*>1). The algorithm will be retrying until the number of generated spheres is *num*. The first iteration tries with the provided porosity, but next iterations increase it if necessary (hence an initialy high porosity can speed-up the algorithm). If *psdSizes* is not defined, *rRelFuzz* (z) and *num* (N) are used so that the porosity given (ρ) is approximately achieved at the end of generation, $r_m = \sqrt[3]{\frac{V(1-\rho)}{\frac{4}{3}\pi(1+z^2)N}}$. The default is ρ=0.5. The optimal value depends on *rRelFuzz* or *psdSizes*.
- **psdSizes** – sieve sizes (particle diameters) when particle size distribution (PSD) is specified
- **psdCumm** – cummulative fractions of particle sizes given by *psdSizes*; must be the same length as *psdSizes* and should be non-decreasing

- **distributeMass** (*bool*) – if `True`, given distribution will be used to distribute sphere's mass rather than radius of them.
- **seed** – number used to initialize the random number generator.

Returns number of created spheres, which can be lower than *num* depending on the method used.

`makeClumpCloud(`*(Vector3)minCorner*, *(Vector3)maxCorner*, *(object)clumps*[, *(bool)periodic=False*[, *(int)num=-1*]]`)` → int

Create random loose packing of clumps within box given by *minCorner* and *maxCorner*. Clumps are selected with equal probability. At most *num* clumps will be positioned if *num* is positive; otherwise, as many clumps as possible will be put in space, until maximum number of attempts to place a new clump randomly is attained.

`particleSD(`*(Vector3)minCorner*, *(Vector3)maxCorner*, *(float)rMean*, *(bool)periodic=False*, *(str)name*, *(int)numSph*[, *(object)radii=[]*[, *(object)passing=[]*[, *(bool)passingIsNotPercentageButCount=False*[, *(int)seed=0*]]]]`)` → int

Create random packing enclosed in box given by minCorner and maxCorner, containing numSph spheres. Returns number of created spheres, which can be < num if the packing is too tight. The computation is done according to the given psd.

`particleSD2(`*(object)radii*, *(object)passing*, *(int)numSph*[, *(bool)periodic=False*[, *(float)cloudPorosity=0.8*[, *(int)seed=0*]]]`)` → int

Create random packing following the given particle size distribution (radii and volume/mass passing for each fraction) and total number of particles *numSph*. The cloud size (periodic or aperiodic) is computed from the PSD and is always cubic.

`particleSD_2d(`*(Vector2)minCorner*, *(Vector2)maxCorner*, *(float)rMean*, *(bool)periodic=False*, *(str)name*, *(int)numSph*[, *(object)radii=[]*[, *(object)passing=[]*[, *(bool)passingIsNotPercentageButCount=False*[, *(int)seed=0*]]]]`)` → int

Create random packing on XY plane, defined by minCorner and maxCorner, containing numSph spheres. Returns number of created spheres, which can be < num if the packing is too tight. The computation is done according to the given psd.

`psd(`[*(int)bins=50*[, *(bool)mass=True*]]`)` → tuple

Return particle size distribution of the packing. :param int bins: number of bins between minimum and maximum diameter :param mass: Compute relative mass rather than relative particle count for each bin. Corresponds to distributeMass parameter for makeCloud. :returns: tuple of `(cumm,edges)`, where `cumm` are cummulative fractions for respective diameters and `edges` are those diameter values. Dimension of both arrays is equal to `bins+1`.

`relDensity()` → float

Relative packing density, measured as sum of spheres' volumes / aabb volume. (Sphere overlaps are ignored.)

`rotate(`*(Vector3)axis*, *(float)angle*`)` → None

Rotate all spheres around packing center (in terms of aabb()), given axis and angle of the rotation.

`save(`*(str)fileName*`)` → None

Save packing to external text file (will be overwritten).

`scale(`*(float)arg2*`)` → None

Scale the packing around its center (in terms of aabb()) by given factor (may be negative).

`toList()` → list

Return packing data as python list.

`toSimulation()`

Append spheres directly to the simulation. In addition calling O.bodies.append, this method also appropriately sets periodic cell information of the simulation.

>>> from yade import pack; from math import * >>> sp=pack.SpherePack()

Create random periodic packing with 20 spheres:

>>> sp.makeCloud((0,0,0),(5,5,5),rMean=.5,rRelFuzz=.5,periodic=True,num=20) 20

Virgin simulation is aperiodic:

>>> O.reset() >>> O.periodic False

Add generated packing to the simulation, rotated by 45° along +z

>>> sp.toSimulation(rot=Quaternion((0,0,1),pi/4),color=(0,0,1)) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]

Periodic properties are transferred to the simulation correctly, including rotation (this could be avoided by explicitly passing "hSize=O.cell.hSize" as an argument):

>>> O.periodic True >>> O.cell.refSize Vector3(5,5,5) >>> O.cell.hSize Matrix3(3.53553,-3.53553,0, 3.53553,3.53553,0, 0,0,5)

The current state (even if rotated) is taken as mechanically undeformed, i.e. with identity transformation:

>>> O.cell.trsf Matrix3(1,0,0, 0,1,0, 0,0,1)

Parameters

- **rot** (*Quaternion/Matrix3*) – rotation of the packing, which will be applied on spheres and will be used to set Cell.trsf as well.
- ****kw** – passed to utils.sphere

Returns list of body ids added (like O.bodies.append)

`translate`(*(Vector3)arg2*) → None
Translate all spheres by given vector.

class `yade._packSpheres.SpherePackIterator`

`__init__`(*(SpherePackIterator)arg2*) → None

`next`() → tuple

Spatial predicates for volumes (defined analytically or by triangulation).

class `yade._packPredicates.Predicate`

`aabb`() → tuple
aabb((Predicate)arg1) → None

`center`() → Vector3

`dim`() → Vector3

class `yade._packPredicates.PredicateBoolean`(*inherits Predicate*)
Boolean operation on 2 predicates (abstract class)

`A`

`B`

`__init__`()
Raises an exception This class cannot be instantiated from Python

class `yade._packPredicates.PredicateDifference`(*inherits PredicateBoolean → Predicate*)
Difference (conjunction with negative predicate) of 2 predicates. A point has to be inside the first and outside the second predicate. Can be constructed using the `-` operator on predicates: `pred1 - pred2`.

`__init__`(*(object)arg2*, *(object)arg3*) → None

class `yade._packPredicates.`**`PredicateIntersection`**(*inherits PredicateBoolean → Predicate*)
Intersection (conjunction) of 2 predicates. A point has to be inside both predicates. Can be constructed using the `&` operator on predicates: `pred1 & pred2`.

`__init__`(*(object)arg2*, *(object)arg3*) → None

class `yade._packPredicates.`**`PredicateSymmetricDifference`**(*inherits PredicateBoolean → Predicate*)
SymmetricDifference (exclusive disjunction) of 2 predicates. A point has to be in exactly one predicate of the two. Can be constructed using the `^` operator on predicates: `pred1 ^ pred2`.

`__init__`(*(object)arg2*, *(object)arg3*) → None

class `yade._packPredicates.`**`PredicateUnion`**(*inherits PredicateBoolean → Predicate*)
Union (non-exclusive disjunction) of 2 predicates. A point has to be inside any of the two predicates to be inside. Can be constructed using the `|` operator on predicates: `pred1 | pred2`.

`__init__`(*(object)arg2*, *(object)arg3*) → None

class `yade._packPredicates.`**`inAlignedBox`**(*inherits Predicate*)
Axis-aligned box predicate

`__init__`(*(Vector3)minAABB*, *(Vector3)maxAABB*) → None
Ctor taking minumum and maximum points of the box (as 3-tuples).

class `yade._packPredicates.`**`inCylinder`**(*inherits Predicate*)
Cylinder predicate

`__init__`(*(Vector3)centerBottom*, *(Vector3)centerTop*, *(float)radius*) → None
Ctor taking centers of the lateral walls (as 3-tuples) and radius.

class `yade._packPredicates.`**`inEllipsoid`**(*inherits Predicate*)
Ellipsoid predicate

`__init__`(*(Vector3)centerPoint*, *(Vector3)abc*) → None
Ctor taking center of the ellipsoid (3-tuple) and its 3 radii (3-tuple).

class `yade._packPredicates.`**`inGtsSurface`**(*inherits Predicate*)
GTS surface predicate

`__init__`(*(object)surface*[, *(bool)noPad*]) → None
Ctor taking a gts.Surface() instance, which must not be modified during instance lifetime. The optional noPad can disable padding (if set to True), which speeds up calls several times. Note: padding checks inclusion of 6 points along +- cardinal directions in the pad distance from given point, which is not exact.

`surf`
The associated gts.Surface object.

class `yade._packPredicates.`**`inHyperboloid`**(*inherits Predicate*)
Hyperboloid predicate

`__init__`(*(Vector3)centerBottom*, *(Vector3)centerTop*, *(float)radius*, *(float)skirt*) → None
Ctor taking centers of the lateral walls (as 3-tuples), radius at bases and skirt (middle radius).

class `yade._packPredicates.`**`inParallelepiped`**(*inherits Predicate*)
Parallelepiped predicate

`__init__`(*(Vector3)o*, *(Vector3)a*, *(Vector3)b*, *(Vector3)c*) → None
Ctor taking four points: `o` (for origin) and then `a`, `b`, `c` which define endpoints of 3 respective edges from `o`.

class `yade._packPredicates.`**`inSphere`**(*inherits Predicate*)
Sphere predicate.

`__init__`(*(Vector3)center*, *(float)radius*) → None
Ctor taking center (as a 3-tuple) and radius

class `yade._packPredicates.`**`notInNotch`**(*inherits Predicate*)
Outside of infinite, rectangle-shaped notch predicate

`__init__`(*(Vector3)centerPoint*, *(Vector3)edge*, *(Vector3)normal*, *(float)aperture*) → None
Ctor taking point in the symmetry plane, vector pointing along the edge, plane normal and aperture size. The side inside the notch is edge×normal. Normal is made perpendicular to the edge. All vectors are normalized at construction time.

Computation of oriented bounding box for cloud of points.

`yade._packObb.cloudBestFitOBB`(*(tuple)arg1*) → tuple
Return (Vector3 center, Vector3 halfSize, Quaternion orientation) of best-fit oriented bounding-box for given tuple of points (uses brute-force velome minimization, do not use for very large clouds).

class `yade._packSpherePadder.SpherePadder`
Geometrical algorithm for filling tetrahedral mesh with spheres; the algorithm was designed by Jean-François Jerier and is described in [Jerier2009].

`__init__`(*(str)fileName*[, *(str)meshType=''*]) → None
Initialize using tetrahedral mesh stored in *fileName*. Type of file is determined by extension: .gmsh implies *meshType*='GMSH', .inp implies *meshType*='INP'. If the extension is different, specify *meshType* explicitly. Possible values are 'GMSH' and 'INP'.

`asSpherePack()` → SpherePack

`densify()` → None

`insert_sphere`(*(float)arg2*, *(float)arg3*, *(float)arg4*, *(float)arg5*) → None

`maxNumberOfSpheres`

`maxOverlapRate`

`maxSolidFractioninProbe`

`meanSolidFraction`

`numberOfSpheres`

`pad_5()` → None

`place_virtual_spheres()` → None

`radiusRange`

`radiusRatio`

`save_mgpost`(*(str)arg2*) → None

`setRadiusRatio`(*(float)arg2*, *(float)arg3*) → None
Like radiusRatio, but taking 2nd parameter.

`virtualRadiusFactor`

8.8 yade.plot module

Module containing utility functions for plotting inside yade. See examples/simple-scene/simple-scene-plot.py or examples/concrete/uniax.py for example of usage.

`yade.plot.data = {}`
Global dictionary containing all data values, common for all plots, in the form {'name':[value,...],...}. Data should be added using plot.addData function. All [value,...] columns have the same length, they are padded with NaN if unspecified.

`yade.plot.plots = {}`
dictionary x-name -> (yspec,...), where yspec is either y-name or (y-name,'line-specification'). If `(yspec,...)` is `None`, then the plot has meaning of image, which will be taken from respective field of plot.imgData.

`yade.plot.labels = {}`
Dictionary converting names in data to human-readable names (TeX names, for instance); if a variable is not specified, it is left untranslated.

`yade.plot.live = False`
Enable/disable live plot updating. Disabled by default for now, since it has a few rough edges.

`yade.plot.liveInterval = 1`
Interval for the live plot updating, in seconds.

`yade.plot.autozoom = True`
Enable/disable automatic plot rezooming after data update.

`yade.plot.plot(`*noShow=False*, *subPlots=True*`)`
Do the actual plot, which is either shown on screen (and nothing is returned: if *noShow* is `False` - note that your yade compilation should present qt4 feature so that figures can be displayed) or, if *noShow* is `True`, returned as matplotlib's Figure object or list of them.

You can use

```
>>> from yade import plot
>>> plot.resetData()
>>> plot.plots={'foo':('bar',)}
>>> plot.plot(noShow=True).savefig('someFile.pdf')
>>> import os
>>> os.path.exists('someFile.pdf')
True
```

to save the figure to file automatically.

Note: For backwards compatibility reasons, *noShow* option will return list of figures for multiple figures but a single figure (rather than list with 1 element) if there is only 1 figure.

`yade.plot.reset()`
Reset all plot-related variables (data, plots, labels)

`yade.plot.resetData()`
Reset all plot data; keep plots and labels intact.

`yade.plot.splitData()`
Make all plots discontinuous at this point (adds nan's to all data fields)

`yade.plot.reverseData()`
Reverse yade.plot.data order.

Useful for tension-compression test, where the initial (zero) state is loaded and, to make data continuous, last part must *end* in the zero state.

`yade.plot.addData(`**d_in*, ***kw*`)`
Add data from arguments name1=value1,name2=value2 to yade.plot.data. (the old {'name1':value1,'name2':value2} is deprecated, but still supported)

New data will be padded with nan's, unspecified data will be nan (nan's don't appear in graphs). This way, equal length of all data is assured so that they can be plotted one against any other.

```
>>> from yade import plot
>>> from pprint import pprint
>>> plot.resetData()
>>> plot.addData(a=1)
>>> plot.addData(b=2)
>>> plot.addData(a=3,b=4)
>>> pprint(plot.data)
{'a': [1, nan, 3], 'b': [nan, 2, 4]}
```

Some sequence types can be given to addData; they will be saved in synthesized columns for individual components.

```
>>> plot.resetData()
>>> plot.addData(c=Vector3(5,6,7),d=Matrix3(8,9,10, 11,12,13, 14,15,16))
>>> pprint(plot.data)
```

```
{'c_x': [5.0],
 'c_y': [6.0],
 'c_z': [7.0],
 'd_xx': [8.0],
 'd_xy': [9.0],
 'd_xz': [10.0],
 'd_yy': [12.0],
 'd_yz': [11.0],
 'd_zx': [14.0],
 'd_zy': [15.0],
 'd_zz': [16.0]}
```

`yade.plot.addAutoData()`

Add data by evaluating contents of plot.plots. Expressions rasing exceptions will be handled gracefully, but warning is printed for each.

```
>>> from yade import plot
>>> from pprint import pprint
>>> O.reset()
>>> plot.resetData()
>>> plot.plots={'O.iter':('O.time',None,'numParticles=len(O.bodies)')}
>>> plot.addAutoData()
>>> pprint(plot.data)
{'O.iter': [0], 'O.time': [0.0], 'numParticles': [0]}
```

Note that each item in plot.plots can be

- an expression to be evaluated (using the `eval` builtin);
- `name=expression` string, where `name` will appear as label in plots, and expression will be evaluated each time;
- a dictionary-like object – current keys are labels of plots and current values are added to plot.data. The contents of the dictionary can change over time, in which case new lines will be created as necessary.

A simple simulation with plot can be written in the following way; note how the energy plot is specified.

```
>>> from yade import plot, utils
>>> plot.plots={'i=O.iter':(O.energy,None,'total energy=O.energy.total()')}
>>> # we create a simple simulation with one ball falling down
>>> plot.resetData()
>>> O.bodies.append(utils.sphere((0,0,0),1))
0
>>> O.dt=utils.PWaveTimeStep()
>>> O.engines=[
...    ForceResetter(),
...    GravityEngine(gravity=(0,0,-10),warnOnce=False),
...    NewtonIntegrator(damping=.4,kinSplit=True),
...    # get data required by plots at every step
...    PyRunner(command='yade.plot.addAutoData()',iterPeriod=1,initRun=True)
... ]
>>> O.trackEnergy=True
>>> O.run(2,True)
>>> pprint(plot.data)
{'gravWork': [0.0, -25.13274...],
 'i': [0, 1],
 'kinRot': [0.0, 0.0],
 'kinTrans': [0.0, 7.5398...],
 'nonviscDamp': [0.0, 10.0530...],
 'total energy': [0.0, -7.5398...]}
```

`yade.plot.saveGnuplot`(*baseName*, *term='wxt'*, *extension=None*, *timestamp=False*, *comment=None*, *title=None*, *varData=False*)

Save data added with plot.addData into (compressed) file and create .gnuplot file that attempts to mimick plots specified with plot.plots.

Parameters

- **baseName** – used for creating baseName.gnuplot (command file for gnuplot), associated `baseName.data.bz2` (data) and output files (if applicable) in the form `baseName.[plot number].extension`
- **term** – specify the gnuplot terminal; defaults to `x11`, in which case gnuplot will draw persistent windows to screen and terminate; other useful terminals are `png`, `cairopdf` and so on
- **extension** – extension for `baseName` defaults to terminal name; fine for png for example; if you use `cairopdf`, you should also say `extension='pdf'` however
- **timestamp** (*bool*) – append numeric time to the basename
- **varData** (*bool*) – whether file to plot will be declared as variable or be in-place in the plot expression
- **comment** – a user comment (may be multiline) that will be embedded in the control file

Returns name of the gnuplot file created.

`yade.plot.saveDataTxt`(*fileName*, *vars=None*)

Save plot data into a (optionally compressed) text file. The first line contains a comment (starting with `#`) giving variable name for each of the columns. This format is suitable for being loaded for further processing (outside yade) with `numpy.genfromtxt` function, which recognizes those variable names (creating numpy array with named entries) and handles decompression transparently.

```
>>> from yade import plot
>>> from pprint import pprint
>>> plot.reset()
>>> plot.addData(a=1,b=11,c=21,d=31)  # add some data here
>>> plot.addData(a=2,b=12,c=22,d=32)
>>> pprint(plot.data)
{'a': [1, 2], 'b': [11, 12], 'c': [21, 22], 'd': [31, 32]}
>>> plot.saveDataTxt('/tmp/dataFile.txt.bz2',vars=('a','b','c'))
>>> import numpy
>>> d=numpy.genfromtxt('/tmp/dataFile.txt.bz2',dtype=None,names=True)
>>> d['a']
array([1, 2])
>>> d['b']
array([11, 12])
```

Parameters

- **fileName** – file to save data to; if it ends with `.bz2` / `.gz`, the file will be compressed using bzip2 / gzip.
- **vars** – Sequence (tuple/list/set) of variable names to be saved. If `None` (default), all variables in plot.plot are saved.

`yade.plot.savePlotSequence`(*fileBase*, *stride=1*, *imgRatio=(5, 7)*, *title=None*, *titleFrames=20*, *lastFrames=30*)

Save sequence of plots, each plot corresponding to one line in history. It is especially meant to be used for utils.makeVideo.

Parameters

- **stride** – only consider every stride-th line of history (default creates one frame per each line)
- **title** – Create title frame, where lines of title are separated with newlines (`\n`) and optional subtitle is separated from title by double newline.

- **titleFrames** (*int*) – Create this number of frames with title (by repeating its filename), determines how long the title will stand in the movie.
- **lastFrames** (*int*) – Repeat the last frame this number of times, so that the movie does not end abruptly.

Returns List of filenames with consecutive frames.

8.9 yade.post2d module

Module for 2d postprocessing, containing classes to project points from 3d to 2d in various ways, providing basic but flexible framework for extracting arbitrary scalar values from bodies/interactions and plotting the results. There are 2 basic components: flatteners and extractors.

The algorithms operate on bodies (default) or interactions, depending on the `intr` parameter of post2d.data.

8.9.1 Flatteners

Instance of classes that convert 3d (model) coordinates to 2d (plot) coordinates. Their interface is defined by the post2d.Flatten class (`__call__`, `planar`, `normal`).

8.9.2 Extractors

Callable objects returning scalar or vector value, given a body/interaction object. If a 3d vector is returned, Flattener.planar is called, which should return only in-plane components of the vector.

8.9.3 Example

This example can be found in examples/concrete/uniax-post.py

```
from yade import post2d
import pylab # the matlab-like interface of matplotlib

O.load('/tmp/uniax-tension.xml.bz2')

# flattener that project to the xz plane
flattener=post2d.AxisFlatten(useRef=False,axis=1)
# return scalar given a Body instance
extractDmg=lambda b: b.state.normDmg
# will call flattener.planar implicitly
# the same as: extractVelocity=lambda b: flattener.planar(b,b.state.vel)
extractVelocity=lambda b: b.state.vel

# create new figure
pylab.figure()
# plot raw damage
post2d.plot(post2d.data(extractDmg,flattener))

# plot smooth damage into new figure
pylab.figure(); ax,map=post2d.plot(post2d.data(extractDmg,flattener,stDev=2e-3))
# show color scale
pylab.colorbar(map,orientation='horizontal')

# raw velocity (vector field) plot
pylab.figure(); post2d.plot(post2d.data(extractVelocity,flattener))

# smooth velocity plot; data are sampled at regular grid
pylab.figure(); ax,map=post2d.plot(post2d.data(extractVelocity,flattener,stDev=1e-3))
```

```
# save last (current) figure to file
pylab.gcf().savefig('/tmp/foo.png')

# show the figures
pylab.show()
```

class `yade.post2d.`**`AxisFlatten`**(*inherits Flatten*)

`__init__`()
:param bool useRef: use reference positions rather than actual positions (only meaningful when operating on Bodies) :param {0,1,2} axis: axis normal to the plane; the return value will be simply position with this component dropped.

`normal`()

`planar`()

class `yade.post2d.`**`CylinderFlatten`**(*inherits Flatten*)
Class for converting 3d point to 2d based on projection onto plane from circle. The y-axis in the projection corresponds to the rotation axis; the x-axis is distance form the axis.

`__init__`()
:param useRef: (bool) use reference positions rather than actual positions :param axis: axis of the cylinder, {0,1,2}

`normal`()

`planar`()

class `yade.post2d.`**`Flatten`**
Abstract class for converting 3d point into 2d. Used by post2d.data2d.

`normal`()
Given position and vector value, return lenght of the vector normal to the flat plane.

`planar`()
Given position and vector value, project the vector value to the flat plane and return its 2 in-plane components.

class `yade.post2d.`**`HelixFlatten`**(*inherits Flatten*)
Class converting 3d point to 2d based on projection from helix. The y-axis in the projection corresponds to the rotation axis

`__init__`()
:param bool useRef: use reference positions rather than actual positions :param (ϑmin,ϑmax) thetaRange: bodies outside this range will be discarded :param float dH_dTheta: inclination of the spiral (per radian) :param {0,1,2} axis: axis of rotation of the spiral :param float periodStart: height of the spiral for zero angle

`normal`()

`planar`()

`yade.post2d.`**`data`**(*extractor*, *flattener*, *intr=False*, *onlyDynamic=True*, *stDev=None*, *relThreshold=3.0*, *perArea=0*, *div=(50, 50)*, *margin=(0, 0)*, *radius=1*)
Filter all bodies/interactions, project them to 2d and extract required scalar value; return either discrete array of positions and values, or smoothed data, depending on whether the stDev value is specified.

The `intr` parameter determines whether we operate on bodies or interactions; the extractor provided should expect to receive body/interaction.

Parameters

- **extractor** (*callable*) – receives Body (or Interaction, if `intr` is `True`) instance, should return scalar, a 2-tuple (vector fields) or None (to skip that body/interaction)

- **flattener** (*callable*) – post2d.Flatten instance, receiving body/interaction, returns its 2d coordinates or `None` (to skip that body/interaction)
- **intr** (*bool*) – operate on interactions rather than bodies
- **onlyDynamic** (*bool*) – skip all non-dynamic bodies
- **stDev** (*float/None*) – standard deviation for averaging, enables smoothing; `None` (default) means raw mode, where discrete points are returned
- **relThreshold** (*float*) – threshold for the gaussian weight function relative to stDev (smooth mode only)
- **perArea** (*int*) – if 1, compute weightedSum/weightedArea rather than weighted average (weightedSum/sumWeights); the first is useful to compute average stress; if 2, compute averages on subdivision elements, not using weight function
- **div** (*(int,int)*) – number of cells for the gaussian grid (smooth mode only)
- **margin** (*(float,float)*) – x,y margins around bounding box for data (smooth mode only)
- **radius** (*float/callable*) – Fallback value for radius (for raw plotting) for non-spherical bodies or interactions; if a callable, receives body/interaction and returns radius

Returns dictionary

Returned dictionary always containing keys 'type' (one of 'rawScalar','rawVector','smoothScalar','smoothVector', depending on value of smooth and on return value from extractor), 'x', 'y', 'bbox'.

Raw data further contains 'radii'.

Scalar fields contain 'val' (value from *extractor*), vector fields have 'valX' and 'valY' (2 components returned by the *extractor*).

`yade.post2d.plot`(*data*, *axes=None*, *alpha=0.5*, *clabel=True*, *cbar=False*, *aspect='equal'*, ***kw*)

Given output from post2d.data, plot the scalar as discrete or smooth plot.

For raw discrete data, plot filled circles with radii of particles, colored by the scalar value.

For smooth discrete data, plot image with optional contours and contour labels.

For vector data (raw or smooth), plot quiver (vector field), with arrows colored by the magnitude.

Parameters

- **axes** – matplotlib.axesinstance where the figure will be plotted; if None, will be created from scratch.
- **data** – value returned by post2d.data
- **clabel** (*bool*) – show contour labels (smooth mode only), or annotate cells with numbers inside (with perArea==2)
- **cbar** (*bool*) – show colorbar (equivalent to calling pylab.colorbar(mappable) on the returned mappable)

Returns tuple of `(axes,mappable)`; mappable can be used in further calls to pylab.colorbar.

8.10 yade.qt module

8.11 yade.timing module

Functions for accessing timing information stored in engines and functors.

See *Timing* section of the programmer's manual, wiki page for some examples.

`yade.timing.reset()`

Zero all timing data.

`yade.timing.stats()`

Print summary table of timing information from engines and functors. Absolute times as well as percentages are given. Sample output:

```
Name                                    Count                 Time        Rel. time
-----------------------------------------------------------------------------------
ForceResetter                       400               9449μs           0.01%
BoundingVolumeMetaEngine            400            1171770μs           1.15%
PersistentSAPCollider               400            9433093μs           9.24%
InteractionGeometryMetaEngine       400           15177607μs          14.87%
InteractionPhysicsMetaEngine        400            9518738μs           9.33%
ConstitutiveLawDispatcher           400           64810867μs          63.49%
  ef2_Spheres_Brefcom_BrefcomLaw
         setup                           4926145            7649131μs           15.25%
         geom                            4926145           23216292μs           46.28%
         material                        4926145            8595686μs           17.14%
         rest                            4926145           10700007μs           21.33%
         TOTAL                                             50161117μs          100.00%
"damper"                            400            1866816μs           1.83%
"strainer"                          400              21589μs           0.02%
"plotDataCollector"                 160              64284μs           0.06%
"damageChecker"                       9               3272μs           0.00%
TOTAL                                            102077490μs         100.00%
```

8.12 yade.utils module

Heap of functions that don't (yet) fit anywhere else.

Devs: please DO NOT ADD more functions here, it is getting too crowded!

`yade.utils.NormalRestitution2DampingRate(`*en*`)`

Compute the normal damping rate as a function of the normal coefficient of restitution e_n. For $e_n \in \langle 0, 1\rangle$ damping rate equals

$$-\frac{\log e_n}{\sqrt{e_n^2 + \pi^2}}$$

`yade.utils.SpherePWaveTimeStep(`*radius, density, young*`)`

Compute P-wave critical timestep for a single (presumably representative) sphere, using formula for P-Wave propagation speed $\Delta t_c = \frac{r}{\sqrt{E/\rho}}$. If you want to compute minimum critical timestep for all spheres in the simulation, use utils.PWaveTimeStep instead.

```
>>> SpherePWaveTimeStep(1e-3,2400,30e9)
2.8284271247461903e-07
```

class `yade.utils.TableParamReader`

Class for reading simulation parameters from text file.

Each parameter is represented by one column, each parameter set by one line. Colums are separated by blanks (no quoting).

First non-empty line contains column titles (without quotes). You may use special column named 'description' to describe this parameter set; if such colum is absent, description will be built by concatenating column names and corresponding values (`param1=34,param2=12.22,param4=foo`)

- from columns ending in `!` (the `!` is not included in the column name)
- from all columns, if no columns end in `!`.

Empty lines within the file are ignored (although counted); `#` starts comment till the end of line. Number of blank-separated columns must be the same for all non-empty lines.

A special value `=` can be used instead of parameter value; value from the previous non-empty line will be used instead (works recursively).

This class is used by utils.readParamsFromTable.

`__init__()`
Setup the reader class, read data into memory.

`paramDict()`
Return dictionary containing data from file given to constructor. Keys are line numbers (which might be non-contiguous and refer to real line numbers that one can see in text editors), values are dictionaries mapping parameter names to their values given in the file. The special value '=' has already been interpreted, `!` (bangs) (if any) were already removed from column titles, `description` column has already been added (if absent).

`yade.utils.`**`aabbDim`**`(`*cutoff=0.0*, *centers=False*`)`
Return dimensions of the axis-aligned bounding box, optionally with relative part *cutoff* cut away.

`yade.utils.`**`aabbExtrema2d`**`(`*pts*`)`
Return 2d bounding box for a sequence of 2-tuples.

`yade.utils.`**`aabbWalls`**`(`*extrema=None*, *thickness=None*, *oversizeFactor=1.5*, ***kw*`)`
Return 6 boxes that will wrap existing packing as walls from all sides; extrema are extremal points of the Aabb of the packing (will be calculated if not specified) thickness is wall thickness (will be 1/10 of the X-dimension if not specified) Walls will be enlarged in their plane by oversizeFactor. returns list of 6 wall Bodies enclosing the packing, in the order minX,maxX,minY,maxY,minZ,maxZ.

`yade.utils.`**`avgNumInteractions`**`(`*cutoff=0.0*, *skipFree=False*`)`
Return average numer of interactions per particle, also known as *coordination number* Z. This number is defined as

$$Z = 2C/N$$

where C is number of contacts and N is number of particles.

With *skipFree*, particles not contributing to stable state of the packing are skipped, following equation (8) given in [Thornton2000]:

$$Z_m = \frac{2C - N_1}{N - N_0 - N_1}$$

Parameters

- **cutoff** – cut some relative part of the sample's bounding box away.
- **skipFree** – see above.

`yade.utils.`**`box`**`(`*center*, *extents*, *orientation=[1, 0, 0, 0]*, *dynamic=None*, *fixed=False*, *wire=False*, *color=None*, *highlight=False*, *material=-1*, *mask=1*`)`
Create box (cuboid) with given parameters.

Parameters extents (*Vector3*) – half-sizes along x,y,z axes

See utils.sphere's documentation for meaning of other parameters.

`yade.utils.`**`chainedCylinder`**`(`*begin=Vector3(0, 0, 0)*, *end=Vector3(1, 0, 0)*, *radius=0.2*, *dynamic=None*, *fixed=False*, *wire=False*, *color=None*, *highlight=False*, *material=-1*, *mask=1*`)`
Create and connect a chainedCylinder with given parameters. The shape generated by repeted calls of this function is the Minkowski sum of polyline and sphere.

Parameters

- **radius** (*Real*) – radius of sphere in the Minkowski sum.

- **begin** (*Vector3*) – first point positioning the line in the Minkowski sum
- **last** (*Vector3*) – last point positioning the line in the Minkowski sum

In order to build a correct chain, last point of element of rank N must correspond to first point of element of rank N+1 in the same chain (with some tolerance, since bounding boxes will be used to create connections.

Returns Body object with the ChainedCylinder shape.

`yade.utils.defaultMaterial()`
Return default material, when creating bodies with utils.sphere and friends, material is unspecified and there is no shared material defined yet. By default, this function returns:

FrictMat(density=1e3,young=1e7,poisson=.3,frictionAngle=.5,label='defaultMat')

`yade.utils.facet`(*vertices*, *dynamic=None*, *fixed=True*, *wire=True*, *color=None*, *highlight=False*, *noBound=False*, *material=-1*, *mask=1*)
Create facet with given parameters.

Parameters

- **vertices** (*[Vector3, Vector3, Vector3]*) – coordinates of vertices in the global coordinate system.
- **wire** (*bool*) – if `True`, facets are shown as skeleton; otherwise facets are filled
- **noBound** (*bool*) – set Body.bounded
- **color** (*Vector3-or-None*) – color of the facet; random color will be assigned if `None`.

See utils.sphere's documentation for meaning of other parameters.

`yade.utils.fractionalBox`(*fraction=1.0*, *minMax=None*)
retrurn (min,max) that is the original minMax box (or aabb of the whole simulation if not specified) linearly scaled around its center to the fraction factor

`yade.utils.loadVars`(*mark=None*)
Load variables from utils.saveVars, which are saved inside the simulation. If `mark==None`, all save variables are loaded. Otherwise only those with the mark passed.

`yade.utils.makeVideo`(*frameSpec*, *out*, *renameNotOverwrite=True*, *fps=24*, *kbps=6000*, *bps=None*)
Create a video from external image files using mencoder. Two-pass encoding using the default mencoder codec (mpeg4) is performed, running multi-threaded with number of threads equal to number of OpenMP threads allocated for Yade.

Parameters

- **frameSpec** – wildcard | sequence of filenames. If list or tuple, filenames to be encoded in given order; otherwise wildcard understood by mencoder's mf:// URI option (shell wildcards such as `/tmp/snap-*.png` or and printf-style pattern like `/tmp/snap-%05d.png`)
- **out** (*str*) – file to save video into
- **renameNotOverwrite** (*bool*) – if True, existing same-named video file will have -*number* appended; will be overwritten otherwise.
- **fps** (*int*) – Frames per second (`-mf fps=…`)
- **kbps** (*int*) – Bitrate (`-lavcopts vbitrate=…`) in kb/s

`yade.utils.perpendicularArea`(*axis*)
Return area perpendicular to given axis (0=x,1=y,2=z) generated by bodies for which the function consider returns True (defaults to returning True always) and which is of the type Sphere.

`yade.utils.plotDirections`(*aabb=()*, *mask=0*, *bins=20*, *numHist=True*, *noShow=False*)
Plot 3 histograms for distribution of interaction directions, in yz,xz and xy planes and (optional but default) histogram of number of interactions per body.

Returns If *noShow* is `False`, displays the figure and returns nothing. If *noShow*, the figure object is returned without being displayed (works the same way as plot.plot).

`yade.utils.plotNumInteractionsHistogram`(*cutoff=0.0*)
Plot histogram with number of interactions per body, optionally cutting away *cutoff* relative axis-aligned box from specimen margin.

`yade.utils.psd`(*bins=5*, *mass=True*, *mask=-1*)
Calculates particle size distribution.

Parameters

- **bins** (*int*) – number of bins
- **mass** (*bool*) – if true, the mass-PSD will be calculated
- **mask** (*int*) – Body.mask for the body

Returns

- binsSizes: list of bin's sizes
- binsProc: how much material (in procents) are in the bin, cumulative
- binsSumCum: how much material (in units) are in the bin, cumulative

binsSizes, binsProc, binsSumCum

`yade.utils.randomColor`()
Return random Vector3 with each component in interval 0…1 (uniform distribution)

`yade.utils.randomizeColors`(*onlyDynamic=False*)
Assign random colors to Shape::color.

If onlyDynamic is true, only dynamic bodies will have the color changed.

`yade.utils.readParamsFromTable`(*tableFileLine=None*, *noTableOk=True*, *unknownOk=False*, ***kw*)
Read parameters from a file and assign them to __builtin__ variables.

The format of the file is as follows (commens starting with # and empty lines allowed):

```
# commented lines allowed anywhere
name1 name2 … # first non-blank line are column headings
                        # empty line is OK, with or without comment
val1    val2    … # 1st parameter set
val2    val2    … # 2nd
…
```

Assigned tags (the `description` column is synthesized if absent,see utils.TableParamReader);

O.tags['description']=… # assigns the description column; might be synthesized O.tags['params']="name1=val1,name2=val2,…" # all explicitly assigned parameters O.tags['defaultParams']="unassignedName1=defaultValue1,…" # parameters that were left at their defaults O.tags['d.id']=O.tags['id']+'.'+O.tags['description'] O.tags['id.d']=O.tags['description']+'.'+O.tags['id']

All parameters (default as well as settable) are saved using utils.saveVars`('table')`.

Parameters

- **tableFile** – text file (with one value per blank-separated columns)
- **tableLine** (*int*) – number of line where to get the values from
- **noTableOk** (*bool*) – if False, raise exception if the file cannot be open; use default values otherwise
- **unknownOk** (*bool*) – do not raise exception if unknown column name is found in the file, and assign it as well

Returns number of assigned parameters `yade.utils.replaceCollider(`*colliderEngine*`)`

Replaces collider (Collider) engine with the engine supplied. Raises error if no collider is in engines.

`yade.utils.runningInBatch()`

Tell whether we are running inside the batch or separately.

`yade.utils.saveVars(`*mark=''*, *loadNow=True*, ***kw*`)`

Save passed variables into the simulation so that it can be recovered when the simulation is loaded again.

For example, variables *a*, *b* and *c* are defined. To save them, use:

```
>>> from yade import utils
>>> utils.saveVars('something',a=1,b=2,c=3)
>>> from yade.params.something import *
>>> a,b,c
(1, 2, 3)
```

those variables will be save in the .xml file, when the simulation itself is saved. To recover those variables once the .xml is loaded again, use

```
>>> utils.loadVars('something')
```

and they will be defined in the yade.params.*mark* module. The *loadNow* parameter calls utils.loadVars after saving automatically.

`yade.utils.sphere(`*center*, *radius*, *dynamic=None*, *fixed=False*, *wire=False*, *color=None*, *highlight=False*, *material=-1*, *mask=1*`)`

Create sphere with given parameters; mass and inertia computed automatically.

Last assigned material is used by default (*material*=-1), and utils.defaultMaterial() will be used if no material is defined at all.

Parameters

- **center** (*Vector3*) – center
- **radius** (*float*) – radius
- **dynamic** (*float*) – deprecated, see "fixed"
- **fixed** (*float*) – generate the body with all DOFs blocked?
- **material** –

  specify Body.material; different types are accepted:

  – int: O.materials[material] will be used; as a special case, if material==-1 and there is no shared materials defined, utils.defaultMaterial() will be assigned to O.materials[0]

  – string: label of an existing material that will be used

  – Material instance: this instance will be used

  – callable: will be called without arguments; returned Material value will be used (Material factory object, if you like)
- **mask** (*int*) – Body.mask for the body
- **wire** – display as wire sphere?
- **highlight** – highlight this body in the viewer?
- **Vector3-or-None** – body's color, as normalized RGB; random color will be assigned if ''None'.

Returns A Body instance with desired characteristics.

Creating default shared material if none exists neither is given:

```
>>> O.reset()
>>> from yade import utils
>>> len(O.materials)
0
>>> s0=utils.sphere([2,0,0],1)
>>> len(O.materials)
1
```

Instance of material can be given:

```
>>> s1=utils.sphere([0,0,0],1,wire=False,color=(0,1,0),material=ElastMat(young=30e9,density=2e3))
>>> s1.shape.wire
False
>>> s1.shape.color
Vector3(0,1,0)
>>> s1.mat.density
2000.0
```

Material can be given by label:

```
>>> O.materials.append(FrictMat(young=10e9,poisson=.11,label='myMaterial'))
1
>>> s2=utils.sphere([0,0,2],1,material='myMaterial')
>>> s2.mat.label
'myMaterial'
>>> s2.mat.poisson
0.11
```

Finally, material can be a callable object (taking no arguments), which returns a Material instance. Use this if you don't call this function directly (for instance, through yade.pack.randomDensePack), passing only 1 *material* parameter, but you don't want material to be shared.

For instance, randomized material properties can be created like this:

```
>>> import random
>>> def matFactory(): return ElastMat(young=1e10*random.random(),density=1e3+1e3*random.random())
...
>>> s3=utils.sphere([0,2,0],1,material=matFactory)
>>> s4=utils.sphere([1,2,0],1,material=matFactory)
```

`yade.utils.`**`trackPerfomance`**`(`*updateTime=5*`)`

Track perfomance of a simulation. (Experimental) Will create new thread to produce some plots. Useful for track perfomance of long run simulations (in bath mode for example).

`yade.utils.`**`typedEngine`**`(`*name*`)`

Return first engine from current O.engines, identified by its type (as string). For example:

```
>>> from yade import utils
>>> O.engines=[InsertionSortCollider(),NewtonIntegrator(),GravityEngine()]
>>> utils.typedEngine("NewtonIntegrator") == O.engines[1]
True
```

`yade.utils.`**`uniaxialTestFeatures`**`(`*filename=None*, *areaSections=10*, *axis=-1*, ***kw*`)`

Get some data about the current packing useful for uniaxial test:

1. Find the dimensions that is the longest (uniaxial loading axis)
2. Find the minimum cross-section area of the specimen by examining several (areaSections) sections perpendicular to axis, computing area of the convex hull for each one. This will work also for non-prismatic specimen.
3. Find the bodies that are on the negative/positive boundary, to which the straining condition should be applied.

Parameters

- **filename** – if given, spheres will be loaded from this file (ASCII format); if not, current simulation will be used.
- **areaSection** (*float*) – number of section that will be used to estimate cross-section
- **axis** (*{0,1,2}*) – if given, force strained axis, rather than computing it from predominant length

Returns dictionary with keys `negIds`, `posIds`, `axis`, `area`.

Warning: The function utils.approxSectionArea uses convex hull algorithm to find the area, but the implementation is reported to be *buggy* (bot works in some cases). Always check this number, or fix the convex hull algorithm (it is documented in the source, see py/_utils.cpp).

`yade.utils.vmData()`

Return memory usage data from Linux's /proc/[pid]/status, line VmData.

`yade.utils.voxelPorosityTriaxial`(*triax*, *resolution=200*, *offset=0*)

Calculate the porosity of a sample, given the TriaxialCompressionEngine.

A function utils.voxelPorosity is invoked, with the volume of a box enclosed by TriaxialCompressionEngine walls. The additional parameter offset allows using a smaller volume inside the box, where each side of the volume is at offset distance from the walls. By this way it is possible to find a more precise porosity of the sample, since at walls' contact the porosity is usually reduced.

A recommended value of offset is bigger or equal to the average radius of spheres inside.

The value of resolution depends on size of spheres used. It can be calibrated by invoking voxelPorosityTriaxial with offset=0 and comparing the result with TriaxialCompressionEngine.porosity. After calibration, the offset can be set to radius, or a bigger value, to get the result.

Parameters

- **triax** – the TriaxialCompressionEngine handle
- **resolution** – voxel grid resolution
- **offset** – offset distance

Returns the porosity of the sample inside given volume

Example invocation:

```
from yade import utils
rAvg=0.03
TriaxialTest(numberOfGrains=200,radiusMean=rAvg).load()
O.dt=-1
O.run(1000)
O.engines[4].porosity
0.44007807740143889
utils.voxelPorosityTriaxial(O.engines[4],200,0)
0.44055412500000002
utils.voxelPorosityTriaxial(O.engines[4],200,rAvg)
0.36798199999999998
```

`yade.utils.waitIfBatch()`

Block the simulation if running inside a batch. Typically used at the end of script so that it does not finish prematurely in batch mode (the execution would be ended in such a case).

`yade.utils.wall`(*position*, *axis*, *sense=0*, *color=None*, *material=-1*, *mask=1*)

Return ready-made wall body.

Parameters

- **position** (*float-or-Vector3*) – center of the wall. If float, it is the position along given axis, the other 2 components being zero

- **axis** (*{0,1,2}*) – orientation of the wall normal (0,1,2) for x,y,z (sc. planes yz, xz, xy)
- **sense** (*{-1,0,1}*) – sense in which to interact (0: both, -1: negative, +1: positive; see Wall)

See utils.sphere's documentation for meaning of other parameters.

`yade.utils.xMirror(`*half*`)`

Mirror a sequence of 2d points around the x axis (changing sign on the y coord). The sequence should start up and then it will wrap from y downwards (or vice versa). If the last point's x coord is zero, it will not be duplicated.

`yade._utils.PWaveTimeStep()` → float

Get timestep accoring to the velocity of P-Wave propagation; computed from sphere radii, rigidities and masses.

`yade._utils.RayleighWaveTimeStep()` → float

Determination of time step according to Rayleigh wave speed of force propagation.

`yade._utils.aabbExtrema(`[*(float)cutoff=0.0*[, *(bool)centers=False*]]`)` → tuple

Return coordinates of box enclosing all bodies

Parameters

- **centers** (*bool*) – do not take sphere radii in account, only their centroids
- **cutoff** (*float ⟨0…1⟩*) – relative dimension by which the box will be cut away at its boundaries.

Returns (lower corner, upper corner) as (Vector3,Vector3)

`yade._utils.approxSectionArea(`*(float)arg1*, *(int)arg2*`)` → float

Compute area of convex hull when when taking (swept) spheres crossing the plane at coord, perpendicular to axis.

`yade._utils.bodyNumInteractionsHistogram(`[*(tuple)aabb*]`)` → tuple

`yade._utils.bodyStressTensors(`[*(bool)revertSign=False*]`)` → list

Compute and return a table with per-particle stress tensors. Each tensor represents the average stress in one particle, obtained from the contour integral of applied load as detailed below. This definition is considering each sphere as a continuum. It can be considered exact in the context of spheres at static equilibrium, interacting at contact points with negligible volume changes of the solid phase (this last assumption is not restricting possible deformations and volume changes at the packing scale).

Proof:

First, we remark the identity: $\sigma_{ij} = \delta_{ij}\sigma_{ij} = x_{i,j}\sigma_{ij} = (x_i\sigma_{ij})_{,j} - x_i\sigma_{ij,j}$.

At equilibrium, the divergence of stress is null: $\sigma_{ij,j} = \mathbf{0}$. Consequently, after divergence theorem: $\frac{1}{V}\int_V \sigma_{ij} dV = \frac{1}{V}\int_V (x_i\sigma_{ij})_{,j} dV = \frac{1}{V}\int_{\partial V} x_i.\sigma_{ij}.\mathbf{n}_j.dS = \frac{1}{V}\sum_k x_i^k.f_j^k$.

The last equality is implicitely based on the representation of external loads as Dirac distributions whose zeros are the so-called *contact points*: 0-sized surfaces on which the *contact forces* are applied, located at x_i in the deformed configuration.

A weighted average of per-body stresses will give the average stress inside the solid phase. There is a simple relation between the stress inside the solid phase and the stress in an equivalent continuum in the absence of fluid pressure. For porosity n, the relation reads: $\sigma_{ij}^{equ.} = (1-n)\sigma_{ij}^{solid}$.

Parameters revertSign (*bool*) – invert the sign of returned tensors components.

`yade._utils.coordsAndDisplacements(`*(int)axis*[, *(tuple)Aabb=()*]`)` → tuple

Return tuple of 2 same-length lists for coordinates and displacements (coordinate minus reference coordinate) along given axis (1st arg); if the Aabb=((x_min,y_min,z_min),(x_max,y_max,z_-max)) box is given, only bodies within this box will be considered.

`yade._utils.createInteraction(`*(int)id1*, *(int)id2*`)` → Interaction

Create interaction between given bodies by hand.

Current engines are searched for IGeomDispatcher and IPhysDispatcher (might be both hidden in InteractionLoop). Geometry is created using `force` parameter of the geometry dispatcher, wherefore the interaction will exist even if bodies do not spatially overlap and the functor would return `false` under normal circumstances.

This function will very likely behave incorrectly for periodic simulations (though it could be extended it to handle it farily easily).

`yade._utils.elasticEnergy(`*(tuple)arg1*`)` → float

`yade._utils.fabricTensor(`[*(bool)splitTensor=False*[, *(bool)revertSign=False*[, *(float)thresholdForce=nan*]]]`)` → tuple

Compute the fabric tensor of the periodic cell. The original paper can be found in [Satake1982].

Parameters

- **splitTensor** (*bool*) – split the fabric tensor into two parts related to the strong and weak contact forces respectively.
- **revertSign** (*bool*) – it must be set to true if the contact law's convention takes compressive forces as positive.
- **thresholdForce** (*Real*) – if the fabric tensor is split into two parts, a threshold value can be specified otherwise the mean contact force is considered by default. It is worth to note that this value has a sign and the user needs to set it according to the convention adopted for the contact law. To note that this value could be set to zero if one wanted to make distinction between compressive and tensile forces.

`yade._utils.flipCell(`[*(Matrix3)flip=Matrix3(0, 0, 0, 0, 0, 0, 0, 0, 0)*]`)` → Matrix3

Flip periodic cell so that angles between R^3 axes and transformed axes are as small as possible. This function relies on the fact that periodic cell defines by repetition or its corners regular grid of points in R^3; however, all cells generating identical grid are equivalent and can be flipped one over another. This necessiatates adjustment of Interaction.cellDist for interactions that cross boundary and didn't before (or vice versa), and re-initialization of collider. The *flip* argument can be used to specify desired flip: integers, each column for one axis; if zero matrix, best fit (minimizing the angles) is computed automatically.

In c++, this function is accessible as `Shop::flipCell`.

This function is currently broken and should not be used.

`yade._utils.forcesOnCoordPlane(`*(float)arg1*, *(int)arg2*`)` → Vector3

`yade._utils.forcesOnPlane(`*(Vector3)planePt*, *(Vector3)normal*`)` → Vector3

Find all interactions deriving from NormShearPhys that cross given plane and sum forces (both normal and shear) on them.

Parameters

- **planePt** (*Vector3*) – a point on the plane
- **normal** (*Vector3*) – plane normal (will be normalized).

`yade._utils.getSpheresMass(`[*(int)mask=-1*]`)` → float

Compute the total mass of spheres in the simulation (might crash for now if dynamic bodies are not spheres), mask parameter is considered

`yade._utils.getSpheresVolume(`[*(int)mask=-1*]`)` → float

Compute the total volume of spheres in the simulation (might crash for now if dynamic bodies are not spheres), mask parameter is considered

`yade._utils.getViscoelasticFromSpheresInteraction(`*(float)tc*, *(float)en*, *(float)es*`)` → dict

Compute viscoelastic interaction parameters from analytical solution of a pair spheres collision problem:

$$k_n = \frac{m}{t_c^2}\left(\pi^2 + (\ln e_n)^2\right)$$
$$c_n = -\frac{2m}{t_c}$$
$$k_t = \frac{2}{7}\frac{m}{t_c^2}\left(\pi^2 + (\ln e_t)^2\right)$$
$$c_t = -\frac{2}{7}\frac{m}{t_c}\ln e_t$$

where k_n, c_n are normal elastic and viscous coefficients and k_t, c_t shear elastic and viscous coefficients. For details see [Pournin2001].

Parameters

- **m** (*float*) – sphere mass m
- **tc** (*float*) – collision time t_c
- **en** (*float*) – normal restitution coefficient e_n
- **es** (*float*) – tangential restitution coefficient e_s

Returns dictionary with keys `kn` (the value of k_n), `cn` (c_n), `kt` (k_t), `ct` (c_t).

`yade._utils.`**`highlightNone`**`()` → None
Reset highlight on all bodies.

`yade._utils.`**`inscribedCircleCenter`**`(`*(Vector3)v1*, *(Vector3)v2*, *(Vector3)v3*`)` → Vector3
Return center of inscribed circle for triangle given by its vertices *v1*, *v2*, *v3*.

`yade._utils.`**`interactionAnglesHistogram`**`(`*(int)axis*[, *(int)mask*[, *(int)bins*[, *(tuple)aabb*]]]`)` → tuple

`yade._utils.`**`kineticEnergy`**`(`[*(bool)findMaxId=False*]`)` → object
Compute overall kinetic energy of the simulation as

$$\sum \frac{1}{2}\left(m_i \mathbf{v}_i^2 + \boldsymbol{\omega}(\mathbf{I}\boldsymbol{\omega}^T)\right).$$

For aspherical bodies, the inertia tensor $\mathbf{I}$ is transformed to global frame, before multiplied by $\boldsymbol{\omega}$, therefore the value should be accurate.

`yade._utils.`**`maxOverlapRatio`**`()` → float
Return maximum overlap ration in interactions (with ScGeom) of two spheres. The ratio is computed as $\frac{u_N}{2(r_1 r_2)/r_1+r_2}$, where u_N is the current overlap distance and r_1, r_2 are radii of the two spheres in contact.

`yade._utils.`**`negPosExtremeIds`**`(`*(int)axis*[, *(float)distFactor*]`)` → tuple
Return list of ids for spheres (only) that are on extremal ends of the specimen along given axis; distFactor multiplies their radius so that sphere that do not touch the boundary coordinate can also be returned.

`yade._utils.`**`normalShearStressTensors`**`(`[*(bool)compressionPositive=False*[, *(bool)splitNormalTensor=False*[, *(float)thresholdForce=nan*]]]`)` → tuple
Compute overall stress tensor of the periodic cell decomposed in 2 parts, one contributed by normal forces, the other by shear forces. The formulation can be found in [Thornton2000], eq. (3):

$$\sigma_{ij} = \frac{2}{V}\sum R N \mathbf{n}_i \mathbf{n}_j + \frac{2}{V}\sum R T \mathbf{n}_i \mathbf{t}_j$$

where V is the cell volume, R is "contact radius" (in our implementation, current distance between particle centroids), $\mathbf{n}$ is the normal vector, $\mathbf{t}$ is a vector perpendicular to $\mathbf{n}$, N and T are norms of normal and shear forces.

Parameters

- **splitNormalTensor** (*bool*) – if true the function returns normal stress tensor split into two parts according to the two subnetworks of strong an weak forces.
- **thresholdForce** (*Real*) – threshold value according to which the normal stress tensor can be split (e.g. a zero value would make distinction between tensile and compressive forces).

`yade._utils.pointInsidePolygon(`*(tuple)arg1*, *(object)arg2*`)` → bool

`yade._utils.porosity(`[*(float)volume=-1*]`)` → float

Compute packing poro sity $\frac{V-V_s}{V}$ where V is overall volume and V_s is volume of spheres.

Parameters volume (*float*) – overall volume which must be specified for aperiodic simulations. For periodic simulations, current volume of the Cell is used.

`yade._utils.ptInAABB(`*(Vector3)arg1*, *(Vector3)arg2*, *(Vector3)arg3*`)` → bool

Return True/False whether the point p is within box given by its min and max corners

`yade._utils.scalarOnColorScale(`*(float)arg1*, *(float)arg2*, *(float)arg3*`)` → Vector3

`yade._utils.setRefSe3()` → None

Set reference positions and orientations of all bodies equal to their current positions and orientations.

`yade._utils.shiftBodies(`*(list)ids*, *(Vector3)shift*`)` → float

Shifts bodies listed in ids without updating their velocities.

`yade._utils.spiralProject(`*(Vector3)pt*, *(float)dH_dTheta*[, *(int)axis=2*[, *(float)periodStart=nan*[, *(float)theta0=0*]]]`)` → tuple

`yade._utils.stressTensorOfPeriodicCell(`[*(bool)smallStrains=False*]`)` → Matrix3

Compute overall (macroscopic) stress of periodic cell using equation published in [Kuhl2001]:

$$\boldsymbol{\sigma} = \frac{1}{V}\sum_c l^c[\mathbf{N}^c f_N^c + \mathbf{T}^{cT}\cdot \mathbf{f}_T^c],$$

where V is volume of the cell, l^c length of interaction c, f_N^c normal force and $\mathbf{f}_T^c$ shear force. Sumed are values over all interactions c. $\mathbf{N}^c$ and $\mathbf{T}^{cT}$ are projection tensors (see the original publication for more details):

$$\mathbf{N} = \mathbf{n}\otimes\mathbf{n} \rightarrow N_{ij} = n_i n_j$$

$$\mathbf{T}^T = \mathbf{I}_{sym}\cdot\mathbf{n} - \mathbf{n}\otimes\mathbf{n}\otimes\mathbf{n} \rightarrow T_{ijk}^T = \frac{1}{2}(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk})n_l - n_i n_j n_k$$

$$\mathbf{T}^T\cdot\mathbf{f}_T \equiv T_{ijk}^T f_k = (\delta_{ik}n_j/2 + \delta_{jk}n_i/2 - n_i n_j n_k)f_k = n_j f_i/2 + n_i f_j/2 - n_i n_j n_k f_k,$$

where n is unit vector oriented along the interaction (normal) and δ is Kronecker's delta. As $\mathbf{n}$ and $\mathbf{f}_T$ are perpendicular (therfore $n_i f_i = 0$) we can write

$$\sigma_{ij} = \frac{1}{V}\sum l[n_i n_j f_N + n_j f_i^T/2 + n_i f_j^T/2]$$

Parameters **smallStrains** (*bool*) – if false (large strains), real values of volume and interaction lengths are computed. If true, only refLength of interactions and initial volume are computed (can save some time).

Returns macroscopic stress tensor as Matrix3

`yade._utils.sumFacetNormalForces(`*(object)ids*[, *(int)axis=-1*]`)` → float
Sum force magnitudes on given bodies (must have shape of the Facet type), considering only part of forces perpendicular to each facet's face; if *axis* has positive value, then the specified axis (0=x, 1=y, 2=z) will be used instead of facet's normals.

`yade._utils.sumForces(`*(list)ids*, *(Vector3)direction*`)` → float
Return summary force on bodies with given *ids*, projected on the *direction* vector.

`yade._utils.sumTorques(`*(list)ids*, *(Vector3)axis*, *(Vector3)axisPt*`)` → float
Sum forces and torques on bodies given in *ids* with respect to axis specified by a point *axisPt* and its direction *axis*.

`yade._utils.totalForceInVolume()` → tuple
Return summed forces on all interactions and average isotropic stiffness, as tuple (Vector3,float)

`yade._utils.unbalancedForce(`[*(bool)useMaxForce=False*]`)` → float
Compute the ratio of mean (or maximum, if *useMaxForce*) summary force on bodies and mean force magnitude on interactions. For perfectly static equilibrium, summary force on all bodies is zero (since forces from interactions cancel out and induce no acceleration of particles); this ratio will tend to zero as simulation stabilizes, though zero is never reached because of finite precision computation. Sufficiently small value can be e.g. 1e-2 or smaller, depending on how much equilibrium it should be.

`yade._utils.voxelPorosity(`[*(int)resolution=200*[, *(Vector3)start=Vector3(0, 0, 0)*[, *(Vector3)end=Vector3(0, 0, 0)*]]]`)` → float
Compute packing porosity $\frac{V-V_v}{V}$ where V is a specified volume (from start to end) and V_v is volume of voxels that fall inside any sphere. The calculation method is to divide whole volume into a dense grid of voxels (at given resolution), and count the voxels that fall inside any of the spheres. This method allows one to calculate porosity in any given sub-volume of a whole sample. It is properly excluding part of a sphere that does not fall inside a specified volume.

Parameters

- **resolution** (*int*) – voxel grid resolution, values bigger than resolution=1600 require a 64 bit operating system, because more than 4GB of RAM is used, a resolution=800 will use 500MB of RAM.
- **start** (*Vector3*) – start corner of the volume.
- **end** (*Vector3*) – end corner of the volume.

`yade._utils.wireAll()` → None
Set Shape::wire on all bodies to True, rendering them with wireframe only.

`yade._utils.wireNoSpheres()` → None
Set Shape::wire to True on non-spherical bodies (Facets, Walls).

`yade._utils.wireNone()` → None
Set Shape::wire on all bodies to False, rendering them as solids.

8.13 yade.ymport module

Import geometry from various formats ('import' is python keyword, hence the name 'ymport').

`yade.ymport.gengeo(`*mntable*, *shift=Vector3(0, 0, 0)*, *scale=1.0*, ***kw*`)`
Imports geometry from LSMGenGeo library and creates spheres.

Parameters

***mntable*: mntable** object, which creates by LSMGenGeo library, see example

***shift*: [float,float,float]** [X,Y,Z] parameter moves the specimen.

***scale*: float** factor scales the given data.

*****kw*: (unused keyword arguments)** is passed to utils.sphere

LSMGenGeo library allows one to create pack of spheres with given [Rmin:Rmax] with null stress inside the specimen. Can be useful for Mining Rock simulation.

Example: examples/regular-sphere-pack/regular-sphere-pack.py, usage of LSMGenGeo library in scripts/test/genCylLSM.py.

- https://answers.launchpad.net/esys-particle/+faq/877
- http://www.access.edu.au/lsmgengeo_python_doc/current/pythonapi/html/GenGeo-module.html
- https://svn.esscc.uq.edu.au/svn/esys3/lsm/contrib/LSMGenGeo/

`yade.ymport.gengeoFile`(*fileName='file.geo', shift=Vector3(0, 0, 0), scale=1.0, orientation=Quaternion((1, 0, 0), 0), **kw*)

Imports geometry from LSMGenGeo .geo file and creates spheres.

Parameters

***filename*: string** file which has 4 colums [x, y, z, radius].

***shift*: Vector3** Vector3(X,Y,Z) parameter moves the specimen.

***scale*: float** factor scales the given data.

***orientation*: quaternion** orientation of the imported geometry

*****kw*: (unused keyword arguments)** is passed to utils.sphere

Returns list of spheres.

LSMGenGeo library allows one to create pack of spheres with given [Rmin:Rmax] with null stress inside the specimen. Can be useful for Mining Rock simulation.

Example: examples/regular-sphere-pack/regular-sphere-pack.py, usage of LSMGenGeo library in scripts/test/genCylLSM.py.

- https://answers.launchpad.net/esys-particle/+faq/877
- http://www.access.edu.au/lsmgengeo_python_doc/current/pythonapi/html/GenGeo-module.html
- https://svn.esscc.uq.edu.au/svn/esys3/lsm/contrib/LSMGenGeo/

`yade.ymport.gmsh`(*meshfile='file.mesh', shift=Vector3(0, 0, 0), scale=1.0, orientation=Quaternion((1, 0, 0), 0), **kw*)

Imports geometry from mesh file and creates facets.

Parameters

***shift*: [float,float,float]** [X,Y,Z] parameter moves the specimen.

***scale*: float** factor scales the given data.

***orientation*: quaternion** orientation of the imported mesh

*****kw*: (unused keyword arguments)** is passed to utils.facet

Returns list of facets forming the specimen.

mesh files can be easily created with GMSH. Example added to examples/regular-sphere-pack/regular-sphere-pack.py

Additional examples of mesh-files can be downloaded from http://www-roc.inria.fr/gamma/download/download.php

`yade.ymport.gts`(*meshfile, shift=(0, 0, 0), scale=1.0, **kw*)

Read given meshfile in gts format.

Parameters

***meshfile*: string** name of the input file.

***shift*: [float,float,float]** [X,Y,Z] parameter moves the specimen.

***scale*: float** factor scales the given data.

*****kw*: (unused keyword arguments)** is passed to utils.facet

Returns list of facets.

`yade.ymport.stl`(*file*, *dynamic=None*, *fixed=True*, *wire=True*, *color=None*, *highlight=False*, *noBound=False*, *material=-1*)

Import geometry from stl file, return list of created facets.

`yade.ymport.text`(*fileName*, *shift=Vector3(0*, *0*, *0)*, *scale=1.0*, ***kw*)

Load sphere coordinates from file, create spheres, insert them to the simulation.

Parameters

***filename*: string** file which has 4 colums [x, y, z, radius].

***shift*: [float,float,float]** [X,Y,Z] parameter moves the specimen.

***scale*: float** factor scales the given data.

*****kw*: (unused keyword arguments)** is passed to utils.sphere

Returns list of spheres.

Lines starting with # are skipped

`yade.ymport.textExt`(*fileName*, *format='x_y_z_r'*, *shift=Vector3(0*, *0*, *0)*, *scale=1.0*, ***kw*)

Load sphere coordinates from file in specific format, create spheres, insert them to the simulation.

Parameters *filename*: string *format*:

the name of output format. Supported *x_y_z_r'(default), 'x_y_z_r_matId*

***shift*: [float,float,float]** [X,Y,Z] parameter moves the specimen.

***scale*: float** factor scales the given data.

*****kw*: (unused keyword arguments)** is passed to utils.sphere

Returns list of spheres.

Lines starting with # are skipped

`yade.ymport.unv`(*fileName*, *shift=(0*, *0*, *0)*, *scale=1.0*, ***kw*)

Import geometry from unv file, return list of created facets.

Parameters

***fileName*: string** name of unv file

***shift*: [float,float,float] | Vector3** [X,Y,Z] parameter moves the specimen.

***scale*: float** factor scales the given data.

*****kw*: (unused keyword arguments)** is passed to utils.facet

unv files are mainly used for FEM analyses (are used by *OOFEM <http://www.oofem.org/>* and *Abakus <http://www.simulia.com/products/abaqus_fea.html>*), but triangular elements can be imported as facets. These files cen be created e.g. with open-source free software *Salome <salome-platform.org>*.

Example: scripts/test/unvRead.py.

Chapter 9

External modules

9.1 miniEigen (math) module

Basic math functions for Yade: small matrix, vector and quaternion classes. This module internally wraps small parts of the Eigen library. Refer to its documentation for details. All classes in this module support pickling.

class `miniEigen.Matrix3`

3x3 float matrix.

Supported operations (`m` is a Matrix3, `f` if a float/int, `v` is a Vector3): `-m`, `m+m`, `m+=m`, `m-m`, `m-=m`, `m*f`, `f*m`, `m*=f`, `m/f`, `m/=f`, `m*m`, `m*=m`, `m*v`, `v*m`, `m==m`, `m!=m`.

`Identity` = **Matrix3(1,0,0, 0,1,0, 0,0,1)**

`Zero` = **Matrix3(0,0,0, 0,0,0, 0,0,0)**

`__init__`() → None

___init___((Matrix3)m) → None

___init___((Quaternion)q) → None

___init___((float)m00, (float)m01, (float)m02, (float)m10, (float)m11, (float)m12, (float)m20, (float)m21, (float)m22) → object

`col`(*(int)arg2*) → Vector3

`determinant`() → float

`diagonal`() → Vector3

`eigenDecomposition`() → tuple

`inverse`() → Matrix3

`polarDecomposition`() → tuple

`row`(*(int)arg2*) → Vector3

`toVoigt`([*(bool)strain=False*]) → Vector6

Convert 2nd order tensor to 6-vector (Voigt notation), symmetrizing the tensor; if *strain* is `True`, multiply non-diagonal compoennts by 2.

`trace`() → float

`transpose`() → Matrix3

class `miniEigen.Matrix6`

6x6 float matrix.

Supported operations (`m` is a Matrix6, `f` if a float/int, `v` is a Vector6): `-m`, `m+m`, `m+=m`, `m-m`, `m-=m`, `m*f`, `f*m`, `m*=f`, `m/f`, `m/=f`, `m*m`, `m*=m`, `m*v`, `v*m`, `m==m`, `m!=m`.

`Identity` = **Matrix6(1,0,0,0,0,0, 0,1,0,0,0,0, 0,0,1,0,0,0, 0,0,0,1,0,0, 0,0,0,0,1,0, 0,0,0,0,0,1)**

`Zero` = **Matrix6(0,0,0,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0,0)**

`__init__``()` → None
 ___init___((Matrix6)m) → None

`col``(`*(int)arg2*`)` → Vector6

`determinant``()` → float

`diagonal``()` → Vector6

`eigenDecomposition``()` → tuple

`inverse``()` → Matrix6

`polarDecomposition``()` → tuple

`row``(`*(int)arg2*`)` → Vector6

`trace``()` → float

`transpose``()` → Matrix6

class `miniEigen.`**`Quaternion`**
 Quaternion representing rotation.

 Supported operations (`q` is a Quaternion, `v` is a Vector3): `q*q` (rotation composition), `q*=q`, `q*v` (rotating `v` by `q`), `q==q`, `q!=q`.

 `Identity` = **Quaternion((1,0,0),0)**

 `Rotate``(`*(Vector3)v*`)` → Vector3

 `__init__``()` → None
  ___init___((Vector3)axis, (float)angle) → object

  ___init___((float)angle, (Vector3)axis) → object

  ___init___((float)w, (float)x, (float)y, (float)z) → None : Initialize from coefficients.

  Note: The order of coefficients is *w*, *x*, *y*, *z*. The [] operator numbers them differently, 0…4 for *x y z w*!

  ___init___((Matrix3)rotMatrix) → None

  ___init___((Quaternion)other) → None

 `conjugate``()` → Quaternion

 `inverse``()` → Quaternion

 `norm``()` → float

 `normalize``()` → None

 `setFromTwoVectors``(`*(Vector3)u*, *(Vector3)v*`)` → Quaternion

 `toAngleAxis``()` → tuple

 `toAxisAngle``()` → tuple

 `toRotationMatrix``()` → Matrix3

class `miniEigen.`**`Vector2`**
 3-dimensional float vector.

 Supported operations (`f` if a float/int, `v` is a Vector3): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`.

 Implicit conversion from sequence (list,tuple, …) of 2 floats.

 `Ones` = **Vector2(1,1)**

 `UnitX` = **Vector2(1,0)**

 `UnitY` = **Vector2(0,1)**

`Zero` = **Vector2(0,0)**

`__init__()` → None
___init___((Vector2)other) → None

___init___((float)x, (float)y) → None

`dot(`*(Vector2)arg2*`)` → float

`norm()` → float

`normalize()` → None

`squaredNorm()` → float

class `miniEigen.Vector2i`
2-dimensional integer vector.

Supported operations (`i` if an int, `v` is a Vector2i): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*i`, `i*v`, `v*=i`, `v==v`, `v!=v`.

Implicit conversion from sequence (list,tuple, …) of 2 integers.

`Ones` = **Vector2i(1,1)**

`UnitX` = **Vector2i(1,0)**

`UnitY` = **Vector2i(0,1)**

`Zero` = **Vector2i(0,0)**

`__init__()` → None
___init___((Vector2i)other) → None

___init___((int)x, (int)y) → None

`dot(`*(Vector2i)arg2*`)` → float

`norm()` → int

`normalize()` → None

`squaredNorm()` → int

class `miniEigen.Vector3`
3-dimensional float vector.

Supported operations (`f` if a float/int, `v` is a Vector3): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`, plus operations with `Matrix3` and `Quaternion`.

Implicit conversion from sequence (list,tuple, …) of 3 floats.

`Ones` = **Vector3(1,1,1)**

`UnitX` = **Vector3(1,0,0)**

`UnitY` = **Vector3(0,1,0)**

`UnitZ` = **Vector3(0,0,1)**

`Zero` = **Vector3(0,0,0)**

`__init__()` → None
___init___((Vector3)other) → None

___init___((float)x, (float)y, (float)z) → None

`cross(`*(Vector3)arg2*`)` → Vector3

`dot(`*(Vector3)arg2*`)` → float

`norm()` → float

`normalize()` → None

`normalized()` → Vector3

`outer(`*(Vector3)arg2*`)` → Matrix3

`squaredNorm()` → float

class `miniEigen.Vector3i`
3-dimensional integer vector.

Supported operations (`i` if an int, `v` is a Vector3i): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*i`, `i*v`, `v*=i`, `v==v`, `v!=v`.

Implicit conversion from sequence (list,tuple, …) of 3 integers.

`Ones` = **Vector3i(1,1,1)**

`UnitX` = **Vector3i(1,0,0)**

`UnitY` = **Vector3i(0,1,0)**

`UnitZ` = **Vector3i(0,0,1)**

`Zero` = **Vector3i(0,0,0)**

`__init__()` → None
___init___((Vector3i)other) → None

___init___((int)x, (int)y, (int)z) → None

`cross(`*(Vector3i)arg2*`)` → Vector3i

`dot(`*(Vector3i)arg2*`)` → float

`norm()` → int

`squaredNorm()` → int

class `miniEigen.Vector6`
6-dimensional float vector.

Supported operations (`f` if a float/int, `v` is a Vector6, `m` is Matrix6): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`, `v*m`, `m*v`.

Implicit conversion from sequence (list,tuple, …) of 6 floats.

`Ones` = **Vector6(1,1,1, 1,1,1)**

`Zero` = **Vector6(0,0,0, 0,0,0)**

`__init__()` → None
___init___((Vector6)other) → None

___init___((float)v0, (float)v1, (float)v2, (float)v3, (float)v4, (float)v5) → object

`head()` → Vector3

`norm()` → float

`normalize()` → None

`normalized()` → Vector6

`squaredNorm()` → float

`tail()` → Vector3

`toSymmTensor(`[*(bool)strain=False*]`)` → Matrix3
Convert Vector6 in the Voigt notation to the corresponding 2nd order symmetric tensor (as Matrix3); if *strain* is `True`, multiply non-diagonal components by .5

class `miniEigen.Vector6i`
6-dimensional float vector.

Supported operations (`f` if a float/int, `v` is a Vector6): `-v`, `v+v`, `v+=v`, `v-v`, `v-=v`, `v*f`, `f*v`, `v*=f`, `v/f`, `v/=f`, `v==v`, `v!=v`.

Implicit conversion from sequence (list,tuple, …) of 6 floats.

`Ones` = **Vector6i(1,1,1, 1,1,1)**

`Zero` = **Vector6i(0,0,0, 0,0,0)**

`__init__`() → None
    ___init___((Vector6i)other) → None

    ___init___((int)v0, (int)v1, (int)v2, (int)v3, (int)v4, (int)v5) → object

`head`() → Vector3i

`norm`() → int

`normalize`() → None

`normalized`() → Vector6i

`squaredNorm`() → int

`tail`() → Vector3i

9.2 gts (GNU Triangulated surface) module

A package for constructing and manipulating triangulated surfaces.

PyGTS is a python binding for the GNU Triangulated Surface (GTS) Library, which may be used to build, manipulate, and perform computations on triangulated surfaces.

The following geometric primitives are provided:

Point - a point in 3D space Vertex - a Point in 3D space that may be used to define a Segment Segment - a line defined by two Vertex end-points Edge - a Segment that may be used to define the edge of a Triangle Triangle - a triangle defined by three Edges Face - a Triangle that may be used to define a face on a Surface Surface - a surface composed of Faces

A tetrahedron is assembled from these primitives as follows. First, create Vertices for each of the tetrahedron's points:

import gts

v1 = gts.Vertex(1,1,1) v2 = gts.Vertex(-1,-1,1) v3 = gts.Vertex(-1,1,-1) v4 = gts.Vertex(1,-1,-1)

Next, connect the four vertices to create six unique Edges:

e1 = gts.Edge(v1,v2) e2 = gts.Edge(v2,v3) e3 = gts.Edge(v3,v1) e4 = gts.Edge(v1,v4) e5 = gts.Edge(v4,v2) e6 = gts.Edge(v4,v3)

The four triangular faces are composed using three edges each:

f1 = gts.Face(e1,e2,e3) f2 = gts.Face(e1,e4,e5) f3 = gts.Face(e2,e5,e6) f4 = gts.Face(e3,e4,e6)

Finally, the surface is assembled from the faces:

s = gts.Surface() for face in [f1,f2,f3,f4]:

    s.add(face)

Some care must be taken in the orientation of the faces. In the above example, the surface normals are pointing inward, and so the surface technically defines a void, rather than a solid. To create a tetrahedron with surface normals pointing outward, use the following instead:

f1.revert() s = Surface() for face in [f1,f2,f3,f4]:

    if not face.is_compatible(s): face.revert()

    s.add(face)

Once the Surface is constructed, there are many different operations that can be performed. For example, the volume can be calculated using:

s.volume()

The difference between two Surfaces s1 and s2 is given by:

s3 = s2.difference(s1)

Etc.

It is also possible to read in GTS data files and plot surfaces to the screen. See the example programs packaged with PyGTS for more information.

class `gts.Edge`(*inherits Segment → Object → object*)

Bases: `gts.Segment`

Edge object

`__init__`

x.__init__(...) initializes x; see help(type(x)) for signature

`belongs_to_tetrahedron()`

Returns True if this Edge e belongs to a tetrahedron. Otherwise False.

Signature: e.belongs_to_tetrahedron()

`contacts()`

Returns number of sets of connected triangles share this Edge e as a contact Edge.

Signature: e.contacts()

`face_number()`

Returns number of faces using this Edge e on Surface s.

Signature: e.face_number(s)

`is_boundary()`

Returns True if this Edge e is a boundary on Surface s. Otherwise False.

Signature: e.is_boundary(s)

`is_ok()`

True if this Edge e is not degenerate or duplicate. False otherwise. Degeneracy implies e.v1.id == e.v2.id.

Signature: e.is_ok()

`is_unattached()`

True if this Edge e is not part of any Triangle.

Signature: e.is_unattached()

class `gts.Face`(*inherits Triangle → Object → object*)

Bases: `gts.Triangle`

Face object

`__init__`

x.__init__(...) initializes x; see help(type(x)) for signature

`is_compatible()`

True if Face f is compatible with all neighbors in Surface s. False otherwise.

Signature: f.is_compatible(s).

`is_ok()`

True if this Face f is non-degenerate and non-duplicate. False otherwise.

Signature: f.is_ok()

`is_on()`

True if this Face f is on Surface s. False otherwise.

Signature: f.is_on(s).

`is_unattached()`

True if this Face f is not part of any Surface.

Signature: f.is_unattached().

neighbor_number()
Returns the number of neighbors of Face f belonging to Surface s.

Signature: f.neighbor_number(s).

neighbors()
Returns a tuple of neighbors of this Face f belonging to Surface s.

Signature: f.neighbors(s).

class gts.**Object**(*inherits object*)
Bases: `object`

Base object

__init__
x.___init___(...) initializes x; see help(type(x)) for signature

id
GTS object id

is_unattached()
True if this Object o is not attached to another Object. Otherwise False.

Trace: o.is_unattached().

class gts.**Point**(*inherits Object → object*)
Bases: `gts.Object`

Point object

__init__
x.___init___(...) initializes x; see help(type(x)) for signature

closest()
Set the coordinates of Point p to the Point on Segment s or Triangle t closest to the Point p2

Signature: p.closest(s,p2) or p.closest(t,p2)

Returns the (modified) Point p.

coords()
Returns a tuple of the x, y, and z coordinates for this Point p.

Signature: p.coords(x,y,z)

distance()
Returns Euclidean distance between this Point p and other Point p2, Segment s, or Triangle t. Signature: p.distance(p2), p.distance(s) or p.distance(t)

distance2()
Returns squared Euclidean distance between Point p and Point p2, Segment s, or Triangle t.

Signature: p.distance2(p2), p.distance2(s), or p.distance2(t)

is_in()
Tests if this Point p is inside or outside Triangle t. The planar projection (x,y) of Point p is tested against the planar projection of Triangle t.

Signature: p.in_circle(p1,p2,p3) or p.in_circle(t)

Returns a +1 if p lies inside, -1 if p lies outside, and 0 if p lies on the triangle.

is_in_circle()
Tests if this Point p is inside or outside circumcircle. The planar projection (x,y) of Point p is tested against the circumcircle defined by the planar projection of p1, p2 and p3, or alternatively the Triangle t

Signature: p.in_circle(p1,p2,p3) or p.in_circle(t)

Returns +1 if p lies inside, -1 if p lies outside, and 0 if p lies on the circle. The Points p1, p2, and p3 must be in counterclockwise order, or the sign of the result will be reversed.

`is_in_rectangle()`
True if this Point p is in box with bottom-left and upper-right Points p1 and p2.

Signature: p.is_in_rectange(p1,p2)

`is_inside()`
True if this Point p is inside or outside Surface s. False otherwise.

Signature: p.in_inside(s)

`is_ok()`
True if this Point p is OK. False otherwise. This method is useful for unit testing and debugging.

Signature: p.is_ok().

`orientation_3d()`
Determines if this Point p is above, below or on plane of 3 Points p1, p2 and p3.

Signature: p.orientation_3d(p1,p2,p3)

Below is defined so that p1, p2 and p3 appear in counterclockwise order when viewed from above the plane.

The return value is positive if p4 lies below the plane, negative if p4 lies above the plane, and zero if the four points are coplanar. The value is an approximation of six times the signed volume of the tetrahedron defined by the four points.

`orientation_3d_sos()`
Determines if this Point p is above, below or on plane of 3 Points p1, p2 and p3.

Signature: p.orientation_3d_sos(p1,p2,p3)

Below is defined so that p1, p2 and p3 appear in counterclockwise order when viewed from above the plane.

The return value is +1 if p4 lies below the plane, and -1 if p4 lies above the plane. Simulation of Simplicity (SoS) is used to break ties when the orientation is degenerate (i.e. the point lies on the plane definedby p1, p2 and p3).

`rotate()`
Rotates Point p around vector dx,dy,dz by angle a. The sense of the rotation is given by the right-hand-rule.

Signature: p.rotate(dx=0,dy=0,dz=0,a=0)

`scale()`
Scales Point p by vector dx,dy,dz.

Signature: p.scale(dx=1,dy=1,dz=1)

`set()`
Sets x, y, and z coordinates of this Point p.

Signature: p.set(x,y,z)

`translate()`
Translates Point p by vector dx,dy,dz.

Signature: p.translate(dx=0,dy=0,dz=0)

`x`
x value

`y`
y value

`z`
z value

class `gts.`**`Segment`**(*inherits Object → object*)
Bases: `gts.Object`

Segment object

`__init__`
x.__init__(...) initializes x; see help(type(x)) for signature

`connects()`
Returns True if this Segment s1 connects Vertices v1 and v2. False otherwise.

Signature: s1.connects(v1,v2).

`intersection()`
Returns the intersection of Segment s with Triangle t

This function is geometrically robust in the sense that it will return None if s and t do not intersect and will return a Vertex if they do. However, the point coordinates are subject to round-off errors. None will be returned if s is contained in the plane defined by t.

Signature: s.intersection(t) or s.intersection(t,boundary).

If boundary is True (default), the boundary of s is taken into account.

Returns a summit of t (if boundary is True), one of the endpoints of s, a new Vertex at the intersection of s with t, or None if s and t don't intersect.

`intersects()`
Checks if this Segment s1 intersects with Segment s2. Returns 1 if they intersect, 0 if an endpoint of one Segment lies on the other Segment, -1 otherwise

Signature: s1.intersects(s2).

`is_ok()`
True if this Segment s is not degenerate or duplicate. False otherwise. Degeneracy implies s.v1.id == s.v2.id.

Signature: s.is_ok().

`midvertex()`
Returns a new Vertex at the mid-point of this Segment s.

Signature: s.midvertex().

`touches()`
Returns True if this Segment s1 touches Segment s2 (i.e., they share a common Vertex). False otherwise.

Signature: s1.touches(s2).

`v1`
Vertex 1

`v2`
Vertex 2

class `gts.`**`Surface`**(*inherits Object → object*)
Bases: `gts.Object`

Surface object

`Nedges`
The number of unique edges

`Nfaces`
The number of unique faces

`Nvertices`
The number of unique vertices

`__init__`
x.__init__(...) initializes x; see help(type(x)) for signature

`add()`
Adds a Face f or Surface s2 to Surface s1.

Signature: s1.add(f) or s2.add(f)

area()
Returns the area of Surface s. The area is taken as the sum of the signed areas of the Faces of s.

Signature: s.area()

boundary()
Returns a tuple of boundary Edges of Surface s.

Signature: s.boundary()

center_of_area()
Returns the coordinates of the center of area of Surface s.

Signature: s.center_of_area()

center_of_mass()
Returns the coordinates of the center of mass of Surface s.

Signature: s.center_of_mass()

cleanup()
Cleans up the Vertices, Edges, and Faces on a Surface s.

Signature: s.cleanup() or s.cleanup(threhold)

If threhold is given, then Vertices that are spaced less than the threshold are merged. Degenerate Edges and Faces are also removed.

coarsen()
Reduces the number of vertices on Surface s.

Signature: s.coarsen(n) and s.coarsen(amin)

n is the smallest number of desired edges (but you may get fewer). amin is the smallest angle between Faces.

copy()
Copys all Faces, Edges and Vertices of Surface s2 to Surface s1.

Signature: s1.copy(s2)

Returns s1.

difference()
Returns the difference of this Surface s1 with Surface s2.

Signature: s1.difference(s2)

distance()
Calculates the distance between the faces of this Surface s1 and the nearest Faces of other s2, and (if applicable) the distance between the boundary of this Surface s1 and the nearest boundary Edges of other s2.

One or two dictionaries are returned (where applicable), the first for the face range and the second for the boundary range. The fields in each dictionary describe statistical results for each population: {min,max,sum,sum2,mean,stddev,n}.

Signature: s1.distance(s2) or s1.distance(s2,delta)

The value delta is a spatial increment defined as the percentage of the diagonal of the bounding box of s2 (default 0.1).

edges()
Returns tuple of Edges on Surface s that have Vertex in list. If a list is not given then all of the Edges are returned.

Signature: s.edges(list) or s.edges()

face_indices()
Returns a tuple of 3-tuples containing Vertex indices for each Face in Surface s. The index for each Vertex in a face corresponds to where it is found in the Vertex tuple vs.

Signature: s.face_indices(vs)

faces()
Returns tuple of Faces on Surface s that have Edge in list. If a list is not given then all of the Faces are returned.

Signature: s.faces(list) s.faces()

fan_oriented()
Returns a tuple of outside Edges of the Faces fanning from Vertex v on this Surface s. The Edges are given in counter-clockwise order.

Signature: s.fan_oriented(v)

intersection()
Returns the intersection of this Surface s1 with Surface s2.

Signature: s1.intersection(s2)

is_closed()
True if Surface s is closed, False otherwise. Note that a closed Surface is also a manifold.

Signature: s.is_closed()

is_manifold()
True if Surface s is a manifold, False otherwise.

Signature: s.is_manifold()

is_ok()
True if this Surface s is OK. False otherwise.

Signature: s.is_ok()

is_orientable()
True if Faces in Surface s have compatible orientation, False otherwise. Note that a closed surface is also a manifold. Note that an orientable surface is also a manifold.

Signature: s.is_orientable()

is_self_intersecting()
Returns True if this Surface s is self-intersecting. False otherwise.

Signature: s.is_self_intersecting()

manifold_faces()
Returns the 2 manifold Faces of Edge e on this Surface s if they exist, or None.

Signature: s.manifold_faces(e)

next
x.next() -> the next value, or raise StopIteration

parent()
Returns Face on this Surface s that has Edge e, or None if the Edge is not on this Surface.

Signature: s.parent(e)

quality_stats()
Returns quality statistics for this Surface f in a dict. The statistics include the {min, max, sum, sum2, mean, stddev, and n} for populations of face_quality, face_area, edge_length, and edge_angle. Each of these names are dictionary keys. See Triangle.quality() for an explanation of the face_quality.

Signature: s.quality_stats()

remove()
Removes Face f from this Surface s.

Signature: s.remove(f)

rotate()
Rotates Surface s about vector dx,dy,dz and angle a. The sense of the rotation is given by the right-hand-rule.

Signature: s.rotate(dx,dy,dz,a)

scale()
Scales Surface s by vector dx,dy,dz.

Signature: s.scale(dx=1,dy=1,dz=1)

split()
Splits a surface into a tuple of connected and manifold components.

Signature: s.split()

stats()
Returns statistics for this Surface f in a dict. The stats include n_faces, n_incompatible_-faces„ n_boundary_edges, n_non_manifold_edges, and the statisics {min, max, sum, sum2, mean, stddev, and n} for populations of edges_per_vertex and faces_per_edge. Each of these names are dictionary keys.

Signature: s.stats()

strip()
Returns a tuple of strips, where each strip is a tuple of Faces that are successive and have one edge in common.

Signature: s.split()

tessellate()
Tessellate each face of this Surface s with 4 triangles. The number of triangles is increased by a factor of 4.

Signature: s.tessellate()

translate()
Translates Surface s by vector dx,dy,dz.

Signature: s.translate(dx=0,dy=0,dz=0)

union()
Returns the union of this Surface s1 with Surface s2.

Signature: s1.union(s2)

vertices()
Returns a tuple containing the vertices of Surface s.

Signature: s.vertices()

volume()
Returns the signed volume of the domain bounded by the Surface s.

Signature: s.volume()

write()
Saves Surface s to File f in GTS ascii format. All the lines beginning with #! are ignored.

Signature: s.write(f)

write_oogl()
Saves Surface s to File f in OOGL (Geomview) format.

Signature: s.write_oogl(f)

write_oogl_boundary()
Saves boundary of Surface s to File f in OOGL (Geomview) format.

Signature: s.write_oogl_boundary(f)

write_vtk()
: Saves Surface s to File f in VTK format.

  Signature: s.write_vtk(f)

class **gts.Triangle**(*inherits Object → object*)
: Bases: gts.Object

  Triangle object

  __init__
  : x.___init___(...) initializes x; see help(type(x)) for signature

  angle()
  : Returns the angle (radians) between Triangles t1 and t2

    Signature: t1.angle(t2)

  area()
  : Returns the area of Triangle t.

    Signature: t.area()

  circumcenter()
  : Returns a Vertex at the center of the circumscribing circle of this Triangle t, or None if the circumscribing circle is not defined.

    Signature: t.circumcircle_center()

  common_edge()
  : Returns Edge common to both this Triangle t1 and other t2. Returns None if the triangles do not share an Edge.

    Signature: t1.common_edge(t2)

  e1
  : Edge 1

  e2
  : Edge 2

  e3
  : Edge 3

  interpolate_height()
  : Returns the height of the plane defined by Triangle t at Point p. Only the x- and y-coordinates of p are considered.

    Signature: t.interpolate_height(p)

  is_compatible()
  : True if this triangle t1 and other t2 are compatible; otherwise False.

    Checks if this triangle t1 and other t2, which share a common Edge, can be part of the same surface without conflict in the surface normal orientation.

    Signature: t1.is_compatible(t2)

  is_ok()
  : True if this Triangle t is non-degenerate and non-duplicate. False otherwise.

    Signature: t.is_ok()

  is_stabbed()
  : Returns the component of this Triangle t that is stabbed by a ray projecting from Point p to z=infinity. The result can be this Triangle t, one of its Edges or Vertices, or None. If the ray is contained in the plan of this Triangle then None is also returned.

    Signature: t.is_stabbed(p)

  normal()
  : Returns a tuple of coordinates of the oriented normal of Triangle t as the cross-product of

two edges, using the left-hand rule. The normal is not normalized. If this triangle is part of a closed and oriented surface, the normal points to the outside of the surface.

Signature: t.normal()

`opposite()`

Returns Vertex opposite to Edge e or Edge opposite to Vertex v for this Triangle t.

Signature: t.opposite(e) or t.opposite(v)

`orientation()`

Determines orientation of the plane (x,y) projection of Triangle t

Signature: t.orientation()

Returns a positive value if Points p1, p2 and p3 in Triangle t appear in counterclockwise order, a negative value if they appear in clockwise order and zero if they are colinear.

`perimeter()`

Returns the perimeter of Triangle t.

Signature: t.perimeter()

`quality()`

Returns the quality of Triangle t.

The quality of a triangle is defined as the ratio of the square root of its surface area to its perimeter relative to this same ratio for an equilateral triangle with the same area. The quality is then one for an equilateral triangle and tends to zero for a very stretched triangle. Signature: t.quality()

`revert()`

Changes the orientation of triangle t, turning it inside out.

Signature: t.revert()

`vertex()`

Returns the Vertex of this Triangle t not in t.e1.

Signature: t.vertex()

`vertices()`

Returns the three oriented set of vertices in Triangle t.

Signature: t.vertices()

class `gts.`**`Vertex`**(*inherits Point → Object → object*)

Bases: `gts.Point`

Vertex object

`__init__`

x.___init___(...) initializes x; see help(type(x)) for signature

`contacts()`

Returns the number of sets of connected Triangles sharing this Vertex v.

Signature: v.contacts().

If sever is True (default: False) and v is a contact vertex then the vertex is replaced in each Triangle with clones.

`encroaches()`

Returns True if this Vertex v is strictly contained in the diametral circle of Edge e. False otherwise.

Only the projection onto the x-y plane is considered.

Signature: v.encroaches(e)

`faces()`

Returns a tuple of Faces that have this Vertex v.

If a Surface s is given, only Vertices on s are considered.

Signature: v.faces() or v.faces(s).

`is_boundary()`
True if this Vertex v is used by a boundary Edge of Surface s.

Signature: v.is_boundary().

`is_connected()`
Return True if this Vertex v1 is connected to Vertex v2 by a Segment.

Signature: v1.is_connected().

`is_ok()`
True if this Vertex v is OK. False otherwise. This method is useful for unit testing and debugging.

Signature: v.is_ok().

`is_unattached()`
True if this Vertex v is not the endpoint of any Segment.

Signature: v.is_unattached().

`neighbors()`
Returns a tuple of Vertices attached to this Vertex v by a Segment.

If a Surface s is given, only Vertices on s are considered.

Signature: v.neighbors() or v.neighbors(s).

`replace()`
Replaces this Vertex v1 with Vertex v2 in all Segments that have v1. Vertex v1 itself is left unchanged.

Signature: v1.replace(v2).

`triangles()`
Returns a list of Triangles that have this Vertex v.

Signature: v.triangles()

Chapter 10

Acknowledging Yade

In order to let users cite Yade consistently in publications, we provide a list of bibliographic references for the different parts of the documentation, as in the citation model pushed by CGAL. 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. A more detailed discussion of the citation model and its application to Yade can be found here.

If new developments are presented and explained in self-contained papers (at the end of a PhD, typically), we will be glad to include them in the documentation and to reference them in the list below. Any other substantial improvement is welcome and can be discussed in the yade-dev mailing list.

10.1 Citing the Yade Project as a whole (the lazy citation method)

If it is not possible to choose the right chapter (but please try), you may cite the documentation [yade:doc] as a whole:

22. Šmilauer, E. Catalano, B. Chareyre, S. Dorofeenko, J. Duriez, A. Gladky, J. Kozicki, C. Modenese, L. Scholtès, L. Sibille, J. Stránský, and K. Thoeni, Yade Documentation (V. Šmilauer, ed.), The Yade Project, 1st ed., 2010. http://yade-dem.org/doc/.

10.2 Citing chapters of Yade Documentation

The first edition of Yade documentation is seen as a collection with the three volumes (or "chapters") listed below, also provided as bibtex entries. Please cite the chapter that is the most relevant in your case. For instance, a paper using one of the documented contact laws will cite the reference documentation [yade:reference]; if programing concepts are discussed, Yade's manual [yade:manual] will be cited; the theoretical background [yade:background] can be used as the refence for contact detection, time-step determination, or periodic boundary conditions.

- **The reference documentation includes details on equations and algorithms found at the highest lev**

  22. Šmilauer, E. Catalano, B. Chareyre, S. Dorofeenko, J. Duriez, A. Gladky, J. Kozicki, C. Modenese, L. Scholtès, L. Sibille, J. Stránský, and K. Thoeni, "Yade Reference Documentation," in Yade Documentation (V. Šmilauer, ed.), The Yade Project, 1st ed., 2010. http://yade-dem.org/doc/.

- **Software design, user's and programmer's manuals are in (pdf version):**

  22. Šmilauer, A. Gladky, J. Kozicki, C. Modenese, and J. Stránský, "Yade Using and Programming," in Yade Documentation (V. Šmilauer, ed.), The Yade Project, 1st ed., 2010. http://yade-dem.org/doc/.

- **Fundamentals of the DEM as implemented in Yade are explained in (pdf version):**

  22. Šmilauer and B. Chareyre, “Yade Dem Formulation”, in Yade Documentation (V. Šmilauer, ed.), The Yade Project, 1st ed., 2010. http://yade-dem.org/doc/formulation.html.

Chapter 11

Publications on Yade

This page should be a relatively complete list of publications on Yade itself or done with Yade. If you publish something, do not hesitate to add it on the list. If PDF is freely available, add url for direct fulltext downlad. If not, consider uploading fulltext in PDF, either to Yade wiki or to other website, if legally permitted.

The last section gives the references that we recommend to use for citing Yade in publications, as explained in the "Acknowledging Yade" section.

Note: This file is generated from doc/yade-articles.bib, doc/yade-conferences.bib, doc/yade-theses.bib and doc/yade-docref.bib.

11.1 Citing Yade

Corresponding bibtex entries here. Pdf versions are available for each of them. See also "Acknowledging Yade".

11.2 Journal articles

11.3 Conference materials

11.4 Master and PhD theses

Chapter 12

References

All external articles referenced in Yade documentation.

Note: This file is generated from doc/references.bib.

Chapter 13

Indices and tables

- *genindex*
- *modindex*
- *search*

Bibliography

[yade:background] V. Šmilauer, B. Chareyre (2010), **Yade dem formulation**. In *Yade Documentation* (V. Šmilauer, ed.), The Yade Project , 1st ed. (fulltext) (http://yade-dem.org/doc/formulation.html)

[yade:doc] V. Šmilauer, E. Catalano, B. Chareyre, S. Dorofeenko, J. Duriez, A. Gladky, J. Kozicki, C. Modenese, L. Scholtès, L. Sibille, J. Stránský, K. Thoeni (2010), **Yade Documentation**. The Yade Project. (http://yade-dem.org/doc/)

[yade:manual] V. Šmilauer, A. Gladky, J. Kozicki, C. Modenese, J. Stránský (2010), **Yade, using and programming**. In *Yade Documentation* (V. Šmilauer, ed.), The Yade Project , 1st ed. (fulltext) (http://yade-dem.org/doc/)

[yade:reference] V. Šmilauer, E. Catalano, B. Chareyre, S. Dorofeenko, J. Duriez, A. Gladky, J. Kozicki, C. Modenese, L. Scholtès, L. Sibille, J. Stránský, K. Thoeni (2010), **Yade Reference Documentation**. In *Yade Documentation* (V. Šmilauer, ed.), The Yade Project , 1st ed. (fulltext) (http://yade-dem.org/doc/)

[Chareyre2011b] Chareyre, B., Cortis, A., Catalano, E., Barthélemy, E. **Pore-scale modeling of viscous flow and induced forces in dense sphere packings**. *Transport in Porous Media* (92), pages 473-493. DOI 10.1007/s11242-011-9915-6 (fulltext)

[Chen2007] Feng Chen, Eric. C. Drumm, Georges Guiochon (2007), **Prediction/verification of particle motion in one dimension with the discrete-element method**. *International Journal of Geomechanics, ASCE* (7), pages 344–352. DOI 10.1061/(ASCE)1532-3641(2007)7:5(344)

[Chen2011] Chen, F., Drumm, E., Guiochon G. (2011), **Coupled discrete element and finite volume solution of two classical soil mechanics problems**. *Computers and Geotechnics*. DOI 10.1016/j.compgeo.2011.03.009 (fulltext)

[Dang2010a] H. K. Dang, M. A. Meguid (2010), **Algorithm to generate a discrete element specimen with predefined properties**. *International Journal of Geomechanics* (10), pages 85-91. DOI 10.1061/(ASCE)GM.1943-5622.0000028

[Dang2010b] Hoang K. Dang, Mohamed A. Meguid (2010), **Evaluating the performance of an explicit dynamic relaxation technique in analyzing non-linear geotechnical engineering problems**. *Computers and Geotechnics* (37), pages 125 - 131. DOI 10.1016/j.compgeo.2009.08.004

[Donze2008] Donzé, F.V. (2008), **Impacts on cohesive frictional geomaterials**. *European Journal of Environmental and Civil Engineering* (12), pages 967–985.

[Duriez2011] J. Duriez, F. Darve, F.-V. Donze (2011), **A discrete modeling-based constitutive relation for infilled rock joints**. *International Journal of Rock Mechanics & Mining Sciences* (48), pages 458–468. DOI 10.1016/j.ijrmms.2010.09.008

[Favier2009] Favier, L., Daudon, D., Donzé, F.V., Mazars, J. (2009), **Predicting the drag coefficient of a granular flow using the discrete element method**. *Journal of Statistical Mechanics: Theory and Experiment* (2009), pages P06012.

[Harthong2009] Harthong, B., Jerier, J. F., Dorémus, P., Imbault, D., Donzé, F. V. (2009), **Modeling of high-density compaction of granular materials by the discrete element method**. *International Journal of Solids and Structures* (46), pages 3357–3364. DOI 10.1016/j.ijsolstr.2009.05.008

[Hassan2010] A. Hassan, B. Chareyre, F. Darve, J. Meyssonier, F. Flin (2010 (submitted)), **Microtomography-based discrete element modelling of creep in snow**. *Granular Matter.*

[Jerier2009] Jerier, Jean-François, Imbault, Didier, Donzé, Fréederic-Victor, Doremus, Pierre (2009), **A geometric algorithm based on tetrahedral meshes to generate a dense polydisperse sphere packing**. *Granular Matter* (11). DOI 10.1007/s10035-008-0116-0

[Jerier2010] Jerier, Jean-François, Richefeu, Vincent, Imbault, Didier, Donzé, Fréderic-Victor (2010), **Packing spherical discrete elements for large scale simulations**. *Computer Methods in Applied Mechanics and Engineering*. DOI 10.1016/j.cma.2010.01.016

[Jerier2010b] J.-F. Jerier, B. Hathong, V. Richefeu, B. Chareyre, D. Imbault, F.-V. Donze, P. Doremus (2010), **Study of cold powder compaction by using the discrete element method**. *Powder Technology* (In Press). DOI 10.1016/j.powtec.2010.08.056

[Kozicki2005a] J. Kozicki (2005), **Discrete lattice model used to describe the fracture process of concrete**. *Discrete Element Group for Risk Mitigation Annual Report 1, Grenoble University of Joseph Fourier, France*, pages 95–101. (fulltext)

[Kozicki2006a] J. Kozicki, J. Tejchman (2006), **2d lattice model for fracture in brittle materials**. *Archives of Hydro-Engineering and Environmental Mechanics* (53), pages 71–88. (fulltext)

[Kozicki2007a] J. Kozicki, J. Tejchman (2007), **Effect of aggregate structure on fracture process in concrete using 2d lattice model"**. *Archives of Mechanics* (59), pages 365–384. (fulltext)

[Kozicki2008] J. Kozicki, F.V. Donzé (2008), **A new open-source software developed for numerical simulations using discrete modeling methods**. *Computer Methods in Applied Mechanics and Engineering* (197), pages 4429–4443. DOI 10.1016/j.cma.2008.05.023 (fulltext)

[Kozicki2009] J. Kozicki, F.V. Donzé (2009), **Yade-open dem: an open-source software using a discrete element method to simulate granular material**. *Engineering Computations* (26), pages 786–805. DOI 10.1108/02644400910985170 (fulltext)

[Lomine2012] Lominé, F., Scholtès, L., Sibille, L., Poullain, P. (2012), **Modelling of fluid-solid interaction in granular media with coupled lb/de methods: application to piping erosion**. *International Journal for Numerical and Analytical Methods in Geomechanics*. DOI 10.1002/nag.1109

[Nicot2011] Nicot, F., Hadda, N., Bourrier, F., Sibille, L., Darve, F. (2011), **Failure mechanisms in granular media: a discrete element analysis**. *Granular Matter* (13), pages 255-260. DOI 10.1007/s10035-010-0242-3

[Nicot2012] Nicot, F., Sibille, L., Darve, F. (2012), **Failure in rate-independent granular materials as a bifurcation toward a dynamic regime**. *International Journal of Plasticity* (29), pages 136-154. DOI 10.1016/j.ijplas.2011.08.002

[Sayeed2011] Sayeed, M.A., Suzuki, K., Rahman, M.M., Mohamad, W.H.W., Razlan, M.A., Ahmad, Z., Thumrongvut, J., Seangatith, S., Sobhan, MA, Mofiz, SA, others (2011), **Strength and deformation characteristics of granular materials under extremely low to high confining pressures in triaxial compression**. *International Journal of Civil & Environmental Engineering IJCEE-IJENS* (11).

[Scholtes2009a] Scholtès, L., Chareyre, B., Nicot, F., Darve, F. (2009), **Micromechanics of granular materials with capillary effects**. *International Journal of Engineering Science* (47), pages 64–75. DOI 10.1016/j.ijengsci.2008.07.002

[Scholtes2009b] Scholtès, L., Hicher, P.-Y., Chareyre, B., Nicot, F., Darve, F. (2009), **On the capillary stress tensor in wet granular materials**. *International Journal for Numerical and Analytical Methods in Geomechanics* (33), pages 1289–1313. DOI 10.1002/nag.767

[Scholtes2009c] Scholtès, L., Chareyre, B., Nicot, F., Darve, F. (2009), **Discrete modelling of capillary mechanisms in multi-phase granular media**. *Computer Modeling in Engineering and Sciences* (52), pages 297–318. DOI ‘ <http://dx.doi.org/>‘_

[Scholtes2010] Scholtès, L., Hicher, P.Y., Sibille, L. (2010), **Multiscale approaches to describe mechanical responses induced by particle removal in granular materials**. *Comptes Rendus Mécanique (CRAS)* (338), pages 627-638. DOI 10.1016/j.crme.2010.10.003

[Scholtes2011] Scholtès, L., Donzé, F.V., Khanal, M. (2011), **Scale effects on strength of geomaterials, case study: coal**. *Journal of the Mechanics and Physics of Solids*.

[Shiu2008] Shiu, W., Donzé, FV, Daudeville, L. (2008), **Compaction process in concrete during missile impact: a dem analysis**. *Computers and Concrete* (5), pages 329–342.

[Shiu2009] Shiu, W., Donze, F.V., Daudeville, L. (2009), **Discrete element modelling of missile impacts on a reinforced concrete target**. *International Journal of Computer Applications in Technology* (34), pages 33–41.

[Smilauer2006] Václav Šmilauer (2006), **The splendors and miseries of yade design**. *Annual Report of Discrete Element Group for Hazard Mitigation.* (fulltext)

[Tejchman2011] Tejchman, J. (2011), **Comparative modelling of shear zone patterns in granular bodies with finite and discrete element model**. *Advances in Bifurcation and Degradation in Geomaterials*, pages 255–260.

[Tran2011] Tran, VT, Donzé, F.V., Marin, P. (2011), **A discrete element model of concrete under high triaxial loading**. *Cement and Concrete Composites.*

[Catalano2009] E. Catalano, B. Chareyre, E. Barthélémy (2009), **Fluid-solid coupling in discrete models**. In *Alert Geomaterials Workshop 2009.*

[Catalano2010a] E. Catalano, B. Chareyre, E. Barthélémy (2010), **A coupled model for fluid-solid interaction analysis in geomaterials**. In *Alert Geomaterials Workshop 2010.*

[Catalano2010b] E. Catalano, B. Chareyre, E. Barthélémy (2010), **Pore scale modelling of stokes flow**. In *GdR MeGe.*

[Catalano2011a] E. Catalano, B. Chareyre, A. Cortis, E. Barthélémy (2011), **A pore-scale hydro-mechanical coupled model for geomaterials**. In *II International Conference on Particle-based Methods - Fundamentals and Applications.* (fulltext)

[Chareyre2009] Chareyre B., Scholtès L. (2009), **Micro-statics and micro-kinematics of capillary phenomena in dense granular materials**. In *POWDERS AND GRAINS 2009: Proceedings of the 6th International Conference on Micromechanics of Granular Media. AIP Conference Proceedings.* DOI 10.1063/1.3180083

[Chareyre2011] B. Chareyre, E. Catalano, E. Barthélémy (2011), **Numerical simulation of hydromechanical couplings by combined discrete element method and finite-volumes**. In *International Conference on Flows and Mechanics in Natural Porous Media from Pore to Field Scale - Pore2Field.* (fulltext)

[Chen2008a] Chen, F., Drumm, E.C., Guiochon, G., Suzuki, K (2008), **Discrete element simulation of 1d upward seepage flow with particle-fluid interaction using coupled open source software**. In *Proceedings of The 12th International Conference of the International Association for Computer Methods and Advances in Geomechanics (IACMAG) Goa, India.*

[Chen2009b] Chen, F., Drumm, E.C., Guiochon, G. (2009), **3d dem analysis of graded rock fill sinkhole repair: particle size effects on the probability of stability**. In *Transportation Research Board Conference (Washington DC).*

[Dang2008a] Dang, H.K., Mohamed, M.A. (2008), **An algorithm to generate a specimen for discrete element simulations with a predefined grain size distribution.**. In *61th Canadian Geotechnical Conference, Edmonton, Alberta.*

[Dang2008b] Dang, H.K., Mohamed, M.A. (2008), **3d simulation of the trap door problem using the discrete element method.**. In *61th Canadian Geotechnical Conference, Edmonton, Alberta.*

[Favier2009] L. Favier, D. Daudon, F. Donzé, J. Mazars (2009), **Validation of a dem granular flow model aimed at forecasting snow avalanche pressure**. In *AIP Conference Proceedings.* DOI 10.1063/1.3180002 (fulltext)

[Gillibert2009] Gillibert L., Flin F., Rolland du Roscoat S., Chareyre B., Philip A., Lesaffre B., Meyssonier J. (2009), **Curvature-driven grain segmentation: application to snow images from x-ray microtomography**. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Miami, USA).*

[Hicher2009] Hicher P.-Y., Scholtès L., Chareyre B., Nicot F., Darve F. (2008), **On the capillary stress tensor in wet granular materials**. In *Inaugural International Conference of the Engineering Mechanics Institute (EM08) - (Minneapolis, USA).*

[Hicher2011] Hicher, P.Y, Scholtès, L., Sibille, L. (2011), **Multiscale modeling of particle removal impact on granular material behavior**. In *Engineering Mechanics Institute, EMI 2011.*

[Kozicki2003a] J. Kozicki, J. Tejchman (2003), **Discrete methods to describe the behaviour of quasi-brittle and granular materials**. In *16th Engineering Mechanics Conference, University of Washington, Seattle, CD–ROM.*

[Kozicki2003c] J. Kozicki, J. Tejchman (2003), **Lattice method to describe the behaviour of quasi-brittle materials**. In *CURE Workshop, Effective use of building materials, Sopot.*

[Kozicki2004a] J. Kozicki, J. Tejchman (2004), **Study of fracture process in concrete using a discrete lattice model**. In *CURE Workshop, Simulations in Urban Engineering, Gdansk.*

[Kozicki2005b] J. Kozicki, J. Tejchman (2005), **Simulations of fracture in concrete elements using a discrete lattice model**. In *Proc. Conf. Computer Methods in Mechanics (CMM 2005), Czestochowa, Poland.*

[Kozicki2005c] J. Kozicki, J. Tejchman (2005), **Simulation of the crack propagation in concrete with a discrete lattice model**. In *Proc. Conf. Analytical Models and New Concepts in Concrete and Masonry Structures (AMCM 2005), Gliwice, Poland.*

[Kozicki2006b] J. Kozicki, J. Tejchman (2006), **Modelling of fracture process in brittle materials using a lattice model**. In *Computational Modelling of Concrete Structures, EURO-C (eds.: G. Meschke, R. de Borst, H. Mang and N. Bicanic), Taylor anf Francis.*

[Kozicki2006c] J. Kozicki, J. Tejchman (2006), **Lattice type fracture model for brittle materials**. In *35th Solid Mechanics Conference (SOLMECH 2006), Krakow.*

[Kozicki2007c] J. Kozicki, J. Tejchman (2007), **Simulations of fracture processes in concrete using a 3d lattice model**. In *Int. Conf. on Computational Fracture and Failure of Materials and Structures (CFRAC 2007), Nantes.* (fulltext)

[Kozicki2007d] J. Kozicki, J. Tejchman (2007), **Effect of aggregate density on fracture process in concrete using 2d discrete lattice model**. In *Proc. Conf. Computer Methods in Mechanics (CMM 2007), Lodz-Spala.*

[Kozicki2007e] J. Kozicki, J. Tejchman (2007), **Modelling of a direct shear test in granular bodies with a continuum and a discrete approach**. In *Proc. Conf. Computer Methods in Mechanics (CMM 2007), Lodz-Spala.*

[Kozicki2007f] J. Kozicki, J. Tejchman (2007), **Investigations of size effect in tensile fracture of concrete using a lattice model**. In *Proc. Conf. Modelling of Heterogeneous Materials with Applications in Construction and Biomedical Engineering (MHM 2007), Prague.*

[Kozicki2011] J. Kozicki, J. Tejchman (2011), **Numerical simulation of sand behaviour using dem with two different descriptions of grain roughness**. In *II International Conference on Particle-based Methods - Fundamentals and Applications.* (fulltext)

[Lomine2010a] Lominé, F., Scholtès, L., Poullain, P., Sibille, L. (2010), **Soil microstructure changes induced by internal fluid flow: investigations with coupled de/lb methods**. In *Proc. of 3rd Euromediterranean Symposium on Advances in Geomaterials and Structures, AGS'10.*

[Lomine2010b] Lominé, F., Poullain, P., Sibille, L. (2010), **Modelling of fluid-solid interaction in granular media with coupled lb/de methods: application to solid particle detachment under hydraulic loading**. In *19th Discrete Simulation of Fluid Dynamics, DSFD 2010.*

[Lomine2011] Lominé, F., Sibille, L., Marot, D. (2011), **A coupled discrete element - lattice botzmann method to investigate internal erosion in soil**. In *Proc. 2nd Int. Symposium on Computational Geomechanics (ComGeo II).*

[Nicot2010] Nicot, F., Sibille, L., Daouadji, A., Hicher, P.Y., Darve, F. (2010), **Multiscale modeling of instability in granular materials**. In *Engineering Mechanics Institute, EMI 2010.*

[Nicot2011] Nicot, F., Hadda, N., Bourrier, F., Sibille, L., Darve, F. (2011), **A discrete element analysis of collapse mechanisms in granular materials**. In *Proc. 2nd Int. Symposium on Computational Geomechanics (ComGeo II).*

[Sari2011] H. Sari, B. Chareyre, E. Catalano, P. Philippe, E. Vincens (2011), **Investigation of internal erosion processes using a coupled dem-fluid method**. In *II International Conference on Particle-based Methods - Fundamentals and Applications.* (fulltext)

[Scholtes2007a] L. Scholtès, B. Chareyre, F. Nicot, F. Darve (2007), **Micromechanical modelling of unsaturated granular media**. In *Proceedings ECCOMAS-MHM07, Prague.*

[Scholtes2008a] L. Scholtès, B. Chareyre, F. Nicot, F. Darve (2008), **Capillary effects modelling in unsaturated granular materials**. In *8th World Congress on Computational Mechanics - 5th European Congress on Computational Methods in Applied Sciences and Engineering, Venice.*

[Scholtes2008b] L. Scholtès, P.-Y. Hicher, F.Nicot, B. Chareyre, F. Darve (2008), **On the capillary stress tensor in unsaturated granular materials**. In *EM08: Inaugural International Conference of the Engineering Mechanics Institute, Minneapolis.*

[Scholtes2009e] Scholtes L, Chareyre B, Darve F (2009), **Micromechanics of partialy saturated granular material**. In *Int. Conf. on Particle Based Methods, ECCOMAS-Particles.*

[Scholtes2011b] L. Scholtès, F. Donzé (2011), **Progressive failure mechanisms in jointed rock: insight from 3d dem modelling**. In *II International Conference on Particle-based Methods - Fundamentals and Applications.* (fulltext)

[Scholtes2011c] Scholtès, L., Hicher, P.Y., Sibille, L. (2011), **A micromechanical approach to describe internal erosion effects in soils**. In *Proc. of Geomechanics and Geotechnics: from micro to macro, IS-Shanghai 2011.*

[Shiu2007a] W. Shiu, F.V. Donze, L. Daudeville (2007), **Discrete element modelling of missile impacts on a reinforced concrete target**. In *Int. Conf. on Computational Fracture and Failure of Materials and Structures (CFRAC 2007), Nantes.*

[Sibille2009] Sibille, L., Scholtès, L. (2009), **Effects of internal erosion on mechanical behaviour of soils: a dem approach**. In *Proc. of International Conference on Particle-Based Methods, Particles 2009.*

[Smilauer2007a] V. Šmilauer (2007), **Discrete and hybrid models: applications to concrete damage**. In *Unpublished.* (fulltext)

[Smilauer2008] Václav Šmilauer (2008), **Commanding c++ with python**. In *ALERT Doctoral school talk.* (fulltext)

[Smilauer2010a] Václav Šmilauer (2010), **Yade: past, present, future**. In *Internal seminary in Laboratoire 3S-R, Grenoble.* (fulltext) (LaTeX sources)

[Stransky2010] Jan Stránský, Milan Jirásek, Václav Šmilauer (2010), **Macroscopic elastic properties of particle models**. In *Proceedings of the International Conference on Modelling and Simulation 2010, Prague.* (fulltext)

[Stransky2011] J. Stransky, M. Jirasek (2011), **Calibration of particle-based models using cells with periodic boundary conditions**. In *II International Conference on Particle-based Methods - Fundamentals and Applications.* (fulltext)

[Thoeni2011] K. Thoeni, C. Lambert, A. Giacomini, S.W. Sloan (2011), **Discrete modelling of a rockfall protective system**. In *Particles 2011: Fundamentals and Applications.* (fulltext)

[Catalano2008a] E. Catalano (2008), **Infiltration effects on a partially saturated slope - an application of the discrete element method and its implementation in the open-source software yade**. Master thesis at *UJF-Grenoble.* (fulltext)

[Chen2009a] Chen, F. (2009), **Coupled flow discrete element method application in granular porous media using open source codes**. PhD thesis at *University of Tennessee, Knoxville.* (fulltext)

[Chen2011] Chen, J. (2011), **Discrete element method (dem) analyses for hot-mix asphalt (hma) mixture compaction**. PhD thesis at *University of Tennessee, Knoxville.*

[Duriez2009a] J. Duriez (2009), **Stabilité des massifs rocheux : une approche mécanique**. PhD thesis at *Institut polytechnique de Grenoble.* (fulltext)

[Favier2009] Favier, L. (2009), **Approche numérique par éléments discrets 3d de la sollicitation d'un écoulement granulaire sur un obstacle**. *Doctoral Dissertations.*

[Jerier2009] Jerier, J.F. (2009), **Modélisation de la compression haute densité des poudres métalliques ductiles par la méthode des éléments discrets (in french)**. PhD thesis at *Université Grenoble I – Joseph Fourier.* (fulltext)

[Kozicki2007b] J. Kozicki (2007), **Application of discrete models to describe the fracture process in brittle materials**. PhD thesis at *Gdansk University of Technology.* (fulltext)

[Marzougui2011] Marzougui, D. (2011), **Hydromechanical modeling of the transport and deformation in bed load sediment with discrete elements and finite volume**. Master thesis at *Ecole Nationale d'Ingénieur de Tunis.* (fulltext)

[Scholtes2009d] Luc Scholtès (2009), **modélisation micromécanique des milieux granulaires partiellement saturés**. PhD thesis at *Institut National Polytechnique de Grenoble.* (fulltext)

[Smilauer2010b] Václav Šmilauer (2010), **Cohesive particle model using the discrete element method on the yade platform**. PhD thesis at *Czech Technical University in Prague, Faculty of Civil Engineering & Université Grenoble I – Joseph Fourier, École doctorale I-MEP2.* (fulltext) (LaTeX sources)

[Smilauer2010c] Václav Šmilauer (2010), **Doctoral thesis statement**. *(PhD thesis summary).* (fulltext) (LaTeX sources)

[Tran2011] Van Tieng TRAN (2011), **Structures en béton soumises à des chargements mécaniques extrêmes: modélisation de la réponse locale par la méthode des éléments discrets (in french)**. PhD thesis at *Université Grenoble I – Joseph Fourier.* (fulltext)

[Addetta2001] G.A. D'Addetta, F. Kun, E. Ramm, H.J. Herrmann (2001), **From solids to granulates - Discrete element simulations of fracture and fragmentation processes in geomaterials.**. In *Continuous and Discontinuous Modelling of Cohesive-Frictional Materials.* (fulltext)

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

[Alonso2004] F. Alonso-Marroquin, R. Garcia-Rojo, H.J. Herrmann (2004), **Micro-mechanical investigation of the granular ratcheting**. In *Cyclic Behaviour of Soils and Liquefaction Phenomena.* (fulltext)

[Bertrand2005] D. Bertrand, F. Nicot, P. Gotteland, S. Lambert (2005), **Modelling a geo-composite cell using discrete analysis**. *Computers and Geotechnics* (32), pages 564–577.

[Bertrand2008] D. Bertrand, F. Nicot, P. Gotteland, S. Lambert (2008), **Discrete element method (dem) numerical modeling of double-twisted hexagonal mesh**. *Canadian Geotechnical Journal* (45), pages 1104–1117.

[Camborde2000a] F. Camborde, C. Mariotti, F.V. Donzé (2000), **Numerical study of rock and concrete behaviour by discrete element modelling**. *Computers and Geotechnics* (27), pages 225–247.

[Chareyre2002a] B. Chareyre, L. Briancon, P. Villard (2002), **Theoretical versus experimental modeling of the anchorage capacity of geotextiles in trenches.**. *Geosynthet. Int.* (9), pages 97–123.

[Chareyre2002b] B. Chareyre, P. Villard (2002), **Discrete element modeling of curved geosynthetic anchorages with known macro-properties.**. In *Proc., First Int. PFC Symposium, Gelsenkirchen, Germany.*

[Chareyre2003] Bruno Chareyre (2003), **Modélisation du comportement d'ouvrages composites sol-géosynthétique par éléments discrets - application aux tranchées d'ancrage en tête de talus.**. PhD thesis at *Grenoble University.* (fulltext)

[Chareyre2005] Bruno Chareyre, Pascal Villard (2005), **Dynamic spar elements and discrete element methods in two dimensions for the modeling of soil-inclusion problems**. *Journal of Engineering Mechanics* (131), pages 689–698. DOI 10.1061/(ASCE)0733-9399(2005)131:7(689) (fulltext)

[CundallStrack1979] P.A. Cundall, O.D.L. Strack (1979), **A discrete numerical model for granular assemblies**. *Geotechnique* (), pages 47–65. DOI 10.1680/geot.1979.29.1.47

[DeghmReport2006] F. V. Donzé (ed.), **Annual report 2006** (2006). *Discrete Element Group for Hazard Mitigation.* Université Joseph Fourier, Grenoble (fulltext)

[Donze1994a] F.V. Donzé, P. Mora, S.A. Magnier (1994), **Numerical simulation of faults and shear zones**. *Geophys. J. Int.* (116), pages 46–52.

[Donze1995a] F.V. Donzé, S.A. Magnier (1995), **Formulation of a three-dimensional numerical model of brittle behavior**. *Geophys. J. Int.* (122), pages 790–802.

[Donze1999a] F.V. Donzé, S.A. Magnier, L. Daudeville, C. Mariotti, L. Davenne (1999), **Study of the behavior of concrete at high strain rate compressions by a discrete element method**. *ASCE J. of Eng. Mech* (125), pages 1154–1163. DOI 10.1016/S0266-352X(00)00013-6

[Donze2004a] F.V. Donzé, P. Bernasconi (2004), **Simulation of the blasting patterns in shaft sinking using a discrete element method**. *Electronic Journal of Geotechnical Engineering* (9), pages 1–44.

[Duriez2010] J. Duriez, F.Darve, F.-V.Donze (2010), **A discrete modeling-based constitutive relation for infilled rock joints**. *International Journal of Rock Mechanics & Mining Sciences.* (in press)

[GarciaRojo2004] R. Garc?a-Rojo, S. McNamara, H. J. Herrmann (2004), **Discrete element methods for the micro-mechanical investigation of granular ratcheting**. In *Proceedings ECCOMAS 2004.* (fulltext)

[Hentz2003] Séebastien Hentz (2003), **Modélisation d'une structure en béton armé soumise ? un choc par la méthode des eléments discrets**. PhD thesis at *Université Grenoble 1 – Joseph Fourier.*

[Hentz2004a] S. Hentz, F.V. Donzé, L.Daudeville (2004), **Discrete element modelling of concrete submitted to dynamic loading at high strain rates**. *Computers and Structures* (82), pages 2509–2524. DOI 10.1016/j.compstruc.2004.05.016

[Hentz2004b] S. Hentz, L. Daudeville, F.V. Donzé (2004), **Identification and validation of a discrete element model for concrete**. *ASCE Journal of Engineering Mechanics* (130), pages 709–719. DOI 10.1061/(ASCE)0733-9399(2004)130:6(709)

[Hentz2005a] S. Hentz, F.V. Donzé, L.Daudeville (2005), **Discrete elements modeling of a reinforced concrete structure submitted to a rock impact**. *Italian Geotechnical Journal* (XXXIX), pages 83–94.

[Hubbard1996] Philip M. Hubbard (1996), **Approximating polyhedra with spheres for time-critical collision detection**. *ACM Trans. Graph.* (15), pages 179–210. DOI 10.1145/231731.231732

[Johnson2008] Scott M. Johnson, John R. Williams, Benjamin K. Cook (2008), **Quaternion-based rigid body rotation integration algorithms for use in particle methods**. *International Journal for Numerical Methods in Engineering* (74), pages 1303–1313. DOI 10.1002/nme.2210

[Jung1997] Derek Jung, Kamal K. Gupta (1997), **Octree-based hierarchical distance maps for collision detection**. *Journal of Robotic Systems* (14), pages 789–806. DOI 10.1002/(SICI)1097-4563(199711)14:11<789::AID-ROB3>3.0.CO;2-Q

[Kettner2011] Lutz Kettner, Andreas Meyer, Afra Zomorodian (2011), **Intersecting sequences of dD iso-oriented boxes**. In *CGAL User and Reference Manual.* (fulltext)

[Klosowski1998] James T. Klosowski, Martin Held, Joseph S. B. Mitchell, Henry Sowizral, Karel Zikan (1998), **Efficient collision detection using bounding volume hierarchies of k-dops**. *IEEE Transactions on Visualization and Computer Graphics* (4), pages 21–36. (fulltext)

[Kuhl2001] E. Kuhl, G. A. D'Addetta, M. Leukart, E. Ramm (2001), **Microplane modelling and particle modelling of cohesive-frictional materials**. In *Continuous and Discontinuous Modelling of Cohesive-Frictional Materials.* DOI 10.1007/3-540-44424-6_3 (fulltext)

[Lu1998] Ya Yan Lu (1998), **Computing the logarithm of a symmetric positive definite matrix**. *Appl. Numer. Math* (26), pages 483–496. DOI 10.1016/S0168-9274(97)00103-7 (fulltext)

[Luding2008] Stefan Luding (2008), **Introduction to discrete element methods**. In *European Journal of Environmental and Civil Engineering.*

[Magnier1998a] S.A. Magnier, F.V. Donzé (1998), **Numerical simulation of impacts using a discrete element method**. *Mech. Cohes.-frict. Mater.* (3), pages 257–276. DOI 10.1002/(SICI)1099-1484(199807)3:3<257::AID-CFM50>3.0.CO;2-Z

[McNamara2008] S. McNamara, R. Garc?a-Rojo, H. J. Herrmann (2008), **Microscopic origin of granular ratcheting**. *Physical Review E* (77). DOI 11.1103/PhysRevE.77.031304

[Munjiza1998] A. Munjiza, K. R. F. Andrews (1998), **Nbs contact detection algorithm for bodies of similar size**. *International Journal for Numerical Methods in Engineering* (43), pages 131–149. DOI 10.1002/(SICI)1097-0207(19980915)43:1<131::AID-NME447>3.0.CO;2-S

[Munjiza2006] A. Munjiza, E. Rougier, N. W. M. John (2006), **Mr linear contact detection algorithm**. *International Journal for Numerical Methods in Engineering* (66), pages 46–71. DOI 10.1002/nme.1538

[Neto2006] Natale Neto, Luca Bellucci (2006), **A new algorithm for rigid body molecular dynamics**. *Chemical Physics* (328), pages 259–268. DOI 10.1016/j.chemphys.2006.07.009

[Omelyan1999] Igor P. Omelyan (1999), **A new leapfrog integrator of rotational motion. the revised angular-momentum approach**. *Molecular Simulation* (22). DOI 10.1080/08927029908022097 (fulltext)

[Pfc3dManual30] ICG (2003), **Pfc3d (particle flow code in 3d) theory and background manual, version 3.0**. Itasca Consulting Group.

[Pion2011] Sylvain Pion, Monique Teillaud (2011), **3D triangulations**. In *CGAL User and Reference Manual.* (fulltext)

[Pournin2001] L. Pournin, Th. M. Liebling, A. Mocellin (2001), **Molecular-dynamics force models for better control of energy dissipation in numerical simulations of dense granular media**. *Phys. Rev. E* (65), pages 011302. DOI 10.1103/PhysRevE.65.011302

[Price2007] Mathew Price, Vasile Murariu, Garry Morrison (2007), **Sphere clump generation and trajectory comparison for real particles**. In *Proceedings of Discrete Element Modelling 2007.* (fulltext)

[Satake1982] M. Satake (1982), **Fabric tensor in granular materials.**. In *Proc., IUTAM Symp. on Deformation and Failure of Granular materials, Delft, The Netherlands.*

[Thornton1991] Colin Thornton, K. K. Yin (1991), **Impact of elastic spheres with and without adhesion**. *Powder technology* (65), pages 153–166. DOI 10.1016/0032-5910(91)80178-L

[Thornton2000] Colin Thornton (2000), **Numerical simulations of deviatoric shear deformation of granular media**. *Géotechnique* (50), pages 43–53. DOI 10.1680/geot.2000.50.1.43

[Verlet1967] Loup Verlet (1967), **Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules**. *Phys. Rev.* (159), pages 98. DOI 10.1103/PhysRev.159.98

[Villard2004a] P. Villard, B. Chareyre (2004), **Design methods for geosynthetic anchor trenches on the basis of true scale experiments and discrete element modelling**. *Canadian Geotechnical Journal* (41), pages 1193–1205.

[Wang2009] Yucang Wang (2009), **A new algorithm to model the dynamics of 3-d bonded rigid bodies with rotations**. *Acta Geotechnica* (4), pages 117–127. DOI 10.1007/s11440-008-0072-1 (fulltext)

[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.

Python Module Index