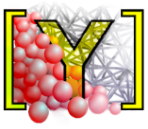


Using and Programming

Václav Šmilauer, Bruno Chareyre, Jérôme Duriez, Alexander Eulitz, Anton Gladky, Ning Guo, Christian Jakob, Janek Kozicki, François Kneib, Chiara Modenese, Jan Stránský, Klaus Thoeni

Yade Documentation 2nd edition, 2015
based on Yade 1.14.0

Citing this document:
Šmilauer V. et al. (2015). Using and Programming. In: *Yade Documentation 2nd ed.* doi:10.5281/zenodo.34043. http://yade-dem.org
See also http://yade-dem/doc/citing.html.

Contents

Chapter 1

Installation

Yade can be installed from packages (pre-compiled binaries) or source code. The choice depends on what you need: if you don't plan to modify Yade itself, package installation is easier. In the contrary case, you must download and install the source code.

1.1 Packages

Pre-built packages are provided for all currently supported Debian and Ubuntu versions of distributions and available on yade-dem.org/packages .

These are `daily` versions of packages and are updating regularly and include all the newly added features.

To install daily-version one needs to add this repository to your /etc/apt/sources.list, add a PGP-key AA915EEB as a trusted and install yadedaily

```
sudo bash -c 'echo "deb http://www.yade-dem.org/packages/ trusty/" >> /etc/apt/sources.list'
wget -O - http://www.yade-dem.org/packages/yadedev_pub.gpg | sudo apt-key add -
sudo apt-get update
sudo apt-get install yadedaily
```

If you have another distribution, not Ubuntu Trusty (Version 14.04 LTS), be sure to use the correct name in the first line (for instance, precise, jessie or wheezy). For the list of currently supported distributions, please visit yade-dem.org/packages.

After that you can normally start Yade using "yadedaily" or "yadedaily-batch" command. yadedaily on older distributions can have some disabled features due to older library versions, shipped with particular distribution.

Git-repository for packaging stuff is available on GitHub. Each branch corresponds to one distribution e.g. precise, jessie etc. The scripts for building all of this stuff is here. It uses pbuilder to build packages, so all packages are building in a clean environment.

If you do not need yadedaily-package any more, just remove the corresponding line in /etc/apt/sources.list and the package itself:

```
sudo apt-get remove yadedaily
```

To remove our key from keyring, execute the following command:

```
sudo apt-key remove AA915EEB
```

Since 2011 all Ubuntu versions (starting from 11.10, Oneiric) and Debian (starting from Wheezy) are having already Yade in their main repositories. There are only stable releases are placed. To install the program, run the following:

```
sudo apt-get install yade
```

To check, what version of Yade is in specific distribution, visit the links for Ubuntu and Debian. Debian-Backports repository is updating regularly to bring the newest Yade to a users of stable Debians.

Daily and stable Yade versions can coexist without any conflicts.

1.2 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.

1.2.1 Download

If you want to install from source, you can install either a release (numbered version, which is frozen) or the current development version (updated by the developers frequently). You should download the development version (called `trunk`) if you want to modify the source code, as you might encounter problems that will be fixed by the developers. Release version will not be modified (except for updates due to critical and easy-to-fix bugs), but they are in a more stabilized state 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 github.

We use GIT (the `git` command) for code management (install the `git` package in your distribution and create a GitHub account):

```
git clone git@github.com:yade/trunk.git
```

will download the whole code repository of `trunk`. Check out *Yade on GitHub* for more.

Alternatively, a read-only checkout is possible via https without a GitHub account (easier if you don't want to modify the main Yade branch):

```
git clone https://github.com/yade/trunk.git
```

For those behind firewall, you can download the sources from our GitHub repository as compressed archive.

Release and trunk sources are compiled in the same way. To be notified about new commits into the trunk, use watch option on GitHub.

1.2.2 Prerequisites

Yade relies on a number of external software to run; they are checked before the compilation starts. Some of them are only optional. The last ones are only relevant for using the fluid coupling module (FlowEngine).

- cmake build system
- gcc compiler (g++); other compilers will not work; you need g++>=4.2 for openMP support
- boost 1.35 or later
- qt4 library
- freeglut3
- libQGLViewer
- python, numpy, ipython
- matplotlib
- eigen3 algebra library (minimal required version 3.2.1)

- gdb debugger
- sqlite3 database engine
- Loki library
- VTK library (optional but recommended)
- CGAL library (optional)
- SuiteSparse sparse algebra library (fluid coupling, optional, requires eigen>=3.1)
- OpenBLAS optimized and parallelized alternative to the standard blas+lapack (fluid coupling, optional)
- Metis matrix preconditioning (fluid coupling, optional)

Most of the list above is very likely already packaged for your distribution. In case you are confronted with some errors concerning not available packages (e.g. Package libmetis-dev is not available) it may be necessary to add yade external ppa from https://launchpad.net/~yade-users/+archive/external:

```
sudo add-apt-repository ppa:yade-users/external
sudo apt-get update
```

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

Warning: If you have Ubuntu 12.10 or older, you need to install libqglviewer-qt4-dev package instead of libqglviewer-dev.

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

```
sudo apt-get install cmake git freeglut3-dev libloki-dev \
libboost-all-dev fakeroot dpkg-dev build-essential g++ \
python-dev ipython python-matplotlib libsqlite3-dev python-numpy python-tk gnuplot \
libgts-dev python-pygraphviz libvtk5-dev python-scientific libeigen3-dev \
python-xlib python-qt4 pyqt4-dev-tools gtk2-engines-pixbuf python-argparse \
libqglviewer-dev python-imaging libjs-jquery python-sphinx python-git python-bibtex \
libxmu-dev libxi-dev libcgal-dev help2man libbz2-dev zlib1g-dev
```

Some of packages (for example, cmake, eigen3) are mandatory, some of them are optional. Watch for notes and warnings/errors, which are shown by cmake during configuration step. If the missing package is optional, some of Yade features will be disabled (see the messages at the end of configuration).

Additional packages, which can become mandatory later:

```
sudo apt-get install python-gts python-minieigen
```

For effective usage of direct solvers in the PFV-type fluid coupling, the following libraries are recommended, together with eigen>=3.1: blas, lapack, suitesparse, and metis. All four of them are available in many different versions. Different combinations are possible and not all of them will work. The following was found to be effective on recent deb-based systems. On ubuntu 12.04, better compile openblas with USE_OPENMP=1, else yade will run on a single core:

```
sudo apt-get install libopenblas-dev libsuitesparse-metis-dev
```

Some packages listed here are relatively new and they can be absent in your distribution (for example, libmetis-dev or python-gts). They can be installed from yade-dem.org/packages or from our external PPA. If not installed the related features will be disabled automatically.

If you are using other distribution, than Debian or its derivatives, you should install the softwares listed above. Their names can differ from the names of Debian-packages.

1.2.3 Compilation

You should create a separate build-place-folder, where Yade will be configured and where the source code will be compiled. Here is an example for a folder structure:

```
myYade/                 ## base directory
        trunk/          ## folder for sourcecode in which you use github
        build/          ## folder in which sources will be compiled; build-directory; use cmake here
        install/        ## install folder
```

Then inside this build-directory you should start cmake to configure the compilation process:

```
cmake -DINSTALL_PREFIX=/path/to/installfolder /path/to/sources
```

For the folder structure given above call the following command in folder "build":

cmake -DINSTALL_PREFIX=../install ../trunk

Additional options can be configured in the same line with the following syntax:

```
cmake -DOPTION1=VALUE1 -DOPTION2=VALUE2
```

The following options are available:

- INSTALL_PREFIX: path where Yade should be installed (/usr/local by default)
- LIBRARY_OUTPUT_PATH: path to install libraries (lib by default)
- DEBUG: compile in debug-mode (OFF by default)
- CMAKE_VERBOSE_MAKEFILE: output additional information during compiling (OFF by default)
- SUFFIX: suffix, added after binary-names (version number by default)
- NOSUFFIX: do not add a suffix after binary-name (OFF by default)
- YADE_VERSION: explicitely set version number (is defined from git-directory by default)
- ENABLE_GUI: enable GUI option (ON by default)
- ENABLE_CGAL: enable CGAL option (ON by default)
- ENABLE_VTK: enable VTK-export option (ON by default)
- ENABLE_OPENMP: enable OpenMP-parallelizing option (ON by default)
- ENABLE_GTS: enable GTS-option (ON by default)
- ENABLE_GL2PS: enable GL2PS-option (ON by default)
- ENABLE_LINSOLV: enable LINSOLV-option (ON by default)
- ENABLE_PFVFLOW: enable PFVFLOW-option, FlowEngine (ON by default)
- runtimePREFIX: used for packaging, when install directory is not the same is runtime directory (/usr/local by default)
- CHUNKSIZE: used, if you want several sources to be compiled at once. Increases compilation speed and RAM-consumption during it (1 by default).

For using an extended parameters of cmake, please, follow the corresponding documentation on cmake-webpage.

If the compilation is finished without errors, you will see all enabled and disabled options. Then start the standard the compilation process:

```
make
```

The compilation process can take a long time, be patient. An additional parameter on many cores systems `-j` can be added to decrease compilation time and split the compilation on many cores. For example, on 4-core machines it would be reasonable to set the parameter `-j4`. Note, the Yade requires approximately 2GB/core for compilation, otherwise the swap-file will be used and a compilation time dramatically increases.

Installing performs with the following command:

```
make install
```

The "install" command will in fact also recompile if source files have been modified. Hence there is no absolute need to type the two commands separately. You may receive make errors if you don't permission to write into the target folder. These errors are not critical but without writing permissions Yade won't be installed in /usr/local/bin/.

After compilation finished successfully the new built can be started by navigating to /path/to/installfolder/bin and calling yade via (based on version yade-2014-02-20.git-a7048f4):

```
cd /path/to/installfolder/bin
./yade-2014-02-20.git-a7048f4
```

For building the documentation you should at first execute the command "make install" and then "make doc" to build it. The generated files will be stored in your current build directory/doc/sphinx/_build. Once again writing permissions are necessary for installing into /usr/local/share/doc/.

"make manpage" command generates and moves manpages in a standard place. "make check" command executes standard test to check the functionality of compiled program.

Yade can be compiled not only by GCC-compiler, but also by CLANG front-end for the LLVM compiler. For that you set the environment variables CC and CXX upon detecting the C and C++ compiler to use:

```
export CC=/usr/bin/clang
export CXX=/usr/bin/clang++
cmake -DOPTION1=VALUE1 -DOPTION2=VALUE2
```

Clang does not support OpenMP-parallelizing for the moment, that is why the feature will be disabled.

1.3 Yubuntu

If you are not running Ubuntu nor Debian, there is a way to create a Yubuntu live-usb on any usb mass-storage device (minimum recommended size is 5GB). It is a way to make a bootable usb-key with a preinstalled minimalist operating system (Xubuntu), including Yadedaily and Paraview.

More informations about this alternative are available here (see the README file first).

Chapter 2

Introduction

2.1 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.

2.1.1 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
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

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

Type `quit()`, `exit()` or simply press `^D` to quit Yade.

[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.

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

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

```
print "Hello world!"
```

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

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

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

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

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

Options:

--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)
--test	Run regression test suite and exit; the exists status is 0 if all tests pass, 1 if a test fails and 2 for an unspecified exception.
--check	Run a series of user-defined check tests as described in /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).

[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.

2.1.2 Creating simulation

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

```
Yade [1]: TriaxialTest(numberOfGrains=200).load()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/usr/lib/x86_64-linux-gnu/yadedaily/py/yade/__init__.pyc in <module>()

NameError: name 'yade' is not defined

Yade [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*.

2.1.3 Running simulation

As explained below, the loop consists in running defined sequence of engines. Step number can be queried by `O.iter` and advancing by one step is done by `O.step()`. Every step advances *virtual time* by current timestep, `O.dt`:

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

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

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

Yade [4]: O.step()

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

Yade [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.

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

Yade [2]: O.pause()

Yade [3]: O.iter
Out[3]: 1809

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

Yade [5]: O.iter
Out[5]: 101809
```

```
Yade [6]: O.stopAtIter=500000

Yade [7]: O.wait()

Yade [8]: O.iter
Out[8]: 101809
```

2.1.4 Saving and loading

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

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

Yade [2]: O.reload()

Yade [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:

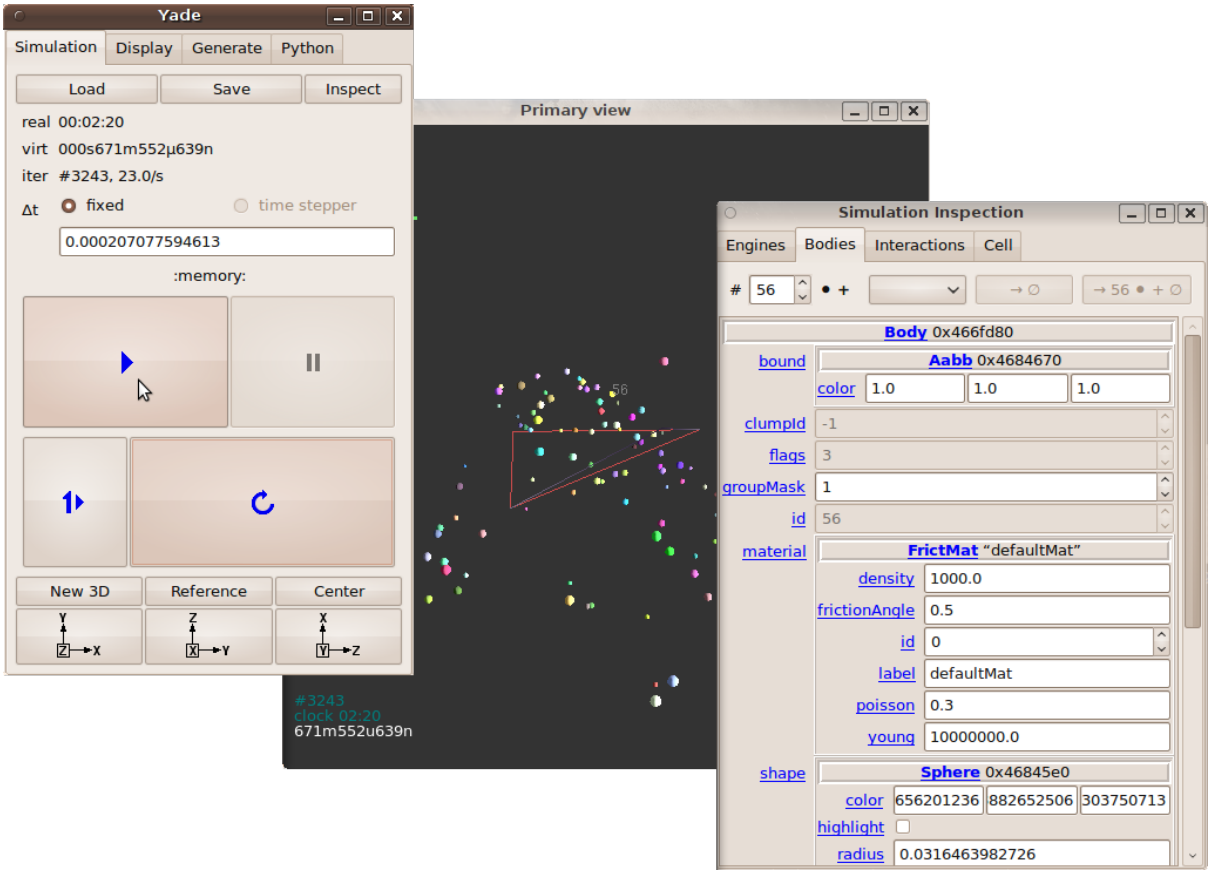
```
Yade [1]: O.saveTmp()

Yade [2]: O.loadTmp()

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

Yade [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.

2.1.5 Graphical interface

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



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

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

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

2.2 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.

2.2.1 Data and functions

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

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

```
Yade [1]: O.bodies       # some data components
Out[1]: <yade.wrapper.BodyContainer at 0x7f2248fb7aa0>

Yade [2]: len(O.bodies)  # there are no bodies as of yet
Out[2]: 0

Yade [3]: O.engines      # functional components, empty at the moment
Out[3]: []
```

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:

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

Yade [2]: s.shape, s.state, s.mat, s.bound
Out[2]:
(<Sphere instance at 0x35817b0>,
 <State instance at 0x35b5d70>,
 <FrictMat instance at 0x365b330>,
 None)

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

Yade [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.

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

Yade [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:

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

Yade [2]: O.bodies[100]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/usr/lib/x86_64-linux-gnu/yadedaily/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:

```
Yade [1]: O.bodies.append(s)
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/usr/lib/x86_64-linux-gnu/yadedaily/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:

```
Yade [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, Cylinder+Sphere etc. Note, however, that it is possible to represent many type of contacts with the basic sphere-sphere geometry (for instance in Ig2_Wall_Sphere_ScGeom).

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

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

```
Yade [1]: O.interactions[0,1]
Out[1]: <Interaction instance at 0x36594f0>

Yade [2]: O.interactions[1,0]     # order of ids is not important
Out[2]: <Interaction instance at 0x36594f0>

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

Yade [4]: i.id1,i.id2
Out[4]: (0, 1)

Yade [5]: i.geom
Out[5]: <ScGeom instance at 0x3659590>

Yade [6]: i.phys
Out[6]: <FrictPhys instance at 0x35e5e40>

Yade [7]: O.interactions[100,10111]
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/usr/lib/x86_64-linux-gnu/yadedaily/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.

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

Yade [2]: O.forces.addF(0,Vector3(1,2,3))
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/usr/lib/x86_64-linux-gnu/yadedaily/py/yade/__init__.pyc in <module>()
----> 1 O.forces.addF(0,Vector3(1,2,3))

NameError: name 'Vector3' is not defined

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

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

Function components

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

- reset forces on bodies from previous step
- approximate collision detection (pass 1)
- detect exact collisions of bodies, update interactions as necessary

- solve interactions, applying forces on bodies
- apply other external conditions (gravity, for instance).
- change position of bodies based on forces, by integrating motion equations.

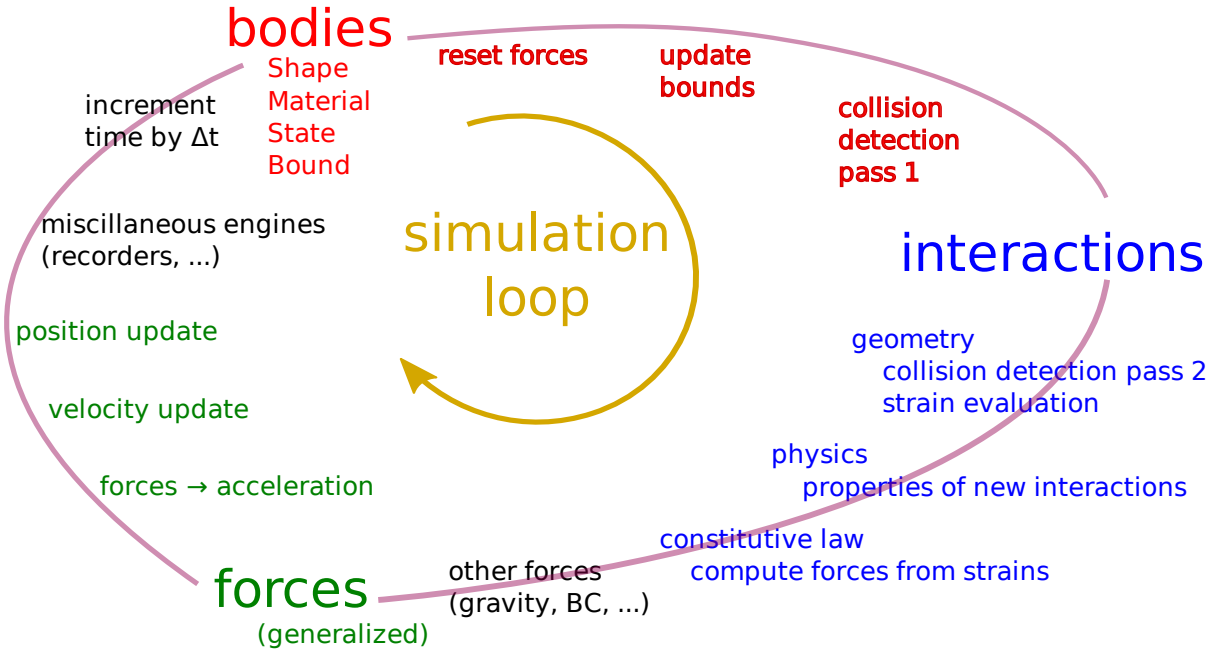


Figure 2.1: Typical simulation loop; each step begins at body-centered bit at 11 o'clock, continues with interaction bit, force application bit, miscillanea and ends with time update.

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

Engines

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

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

There are 3 fundamental types of Engines:

GlobalEngines operating on the whole simulation (e.g. 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_ScGeom(),Ig2_Facet_Sphere_ScGeom()],
        [Ip2_FrictMat_FrictMat_FrictPhys()],
        [Law2_ScGeom_FrictPhys_CundallStrack()],
),
```

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

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

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

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.

Note: When Yade starts, O.engines is filled with a reasonable default list, so that it is not strictly necessary to redefine it when trying simple things. The default scene will handle spheres, boxes, and

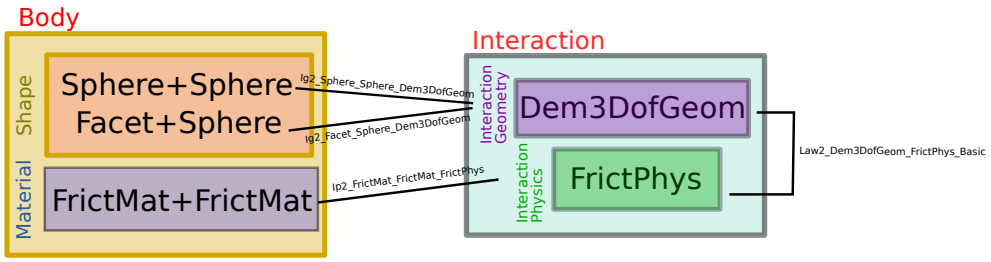


Figure 2.2: Chain of functors producing and accepting certain types. In the case shown, the `Ig2` functors produce ScfGeom instances from all handled Shape combinations; the `Ig2` functor produces FrictMat. The constitutive law functor `Law2` accepts the combination of types produced. Note that the types are stated in the functor's class names.

facets with frictional properties correctly, and adjusts the timestep dynamically. You can find an example in simple-scene-default-engines.py.

Chapter 3

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).

3.1 Introduction

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

3.2 Hands-on

3.2.1 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.

3.2.2 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:

```
Yade [6]: (1+3*4)**2        # usual rules for operator precedence, ** is exponentiation
Out[6]: 169

Yade [7]: import math       # gain access to "module" of functions

Yade [8]: math.sqrt(2)      # use a function from that module
Out[8]: 1.4142135623730951
```

```
Yade [9]: import math as m  # use the module under a different name

Yade [10]: m.cos(m.pi)
Out[10]: -1.0

Yade [11]: from math import *  # import everything so that it can be used without module name

Yade [12]: cos(pi)
Out[12]: -1.0
```

Variables:

```
Yade [13]: a=1; b,c=2,3       # multiple commands separated with ;, multiple assignment

Yade [14]: a+b+c
Out[14]: 6
```

Sequences

Lists

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

```
Yade [15]: a=[1,2,3]          # list of numbers

Yade [16]: a[0]               # first element has index 0
Out[16]: 1

Yade [17]: a[-1]              # negative counts from the end
Out[17]: 3

Yade [18]: a[3]               # error
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/usr/lib/x86_64-linux-gnu/yadedaily/py/yade/__init__.pyc in <module>()
----> 1 a[3]               # error

IndexError: list index out of range

Yade [19]: len(a)             # number of elements
Out[19]: 3

Yade [20]: a[1:]              # from second element to the end
Out[20]: [2, 3]

Yade [21]: a+=[4,5]           # extend the list

Yade [22]: a+=[6]; a.append(7) # extend with single value, both have the same effect

Yade [23]: 9 in a             # test presence of an element
Out[23]: False
```

Lists can be created in various ways:

```
Yade [24]: range(10)
Out[24]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Yade [25]: range(10)[-1]
Out[25]: 9
```

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

```
Yade [26]: [a**2 for a in range(20) if a%2==0]
Out[26]: [0, 4, 16, 36, 64, 100, 144, 196, 256, 324]
```

Tuples

Tuples are constant sequences:

```
Yade [27]: b=(1,2,3)

Yade [28]: b[0]
Out[28]: 1

Yade [29]: b[0]=4              # error
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/usr/lib/x86_64-linux-gnu/yadedaily/py/yade/__init__.pyc in <module>()
----> 1 b[0]=4              # error

TypeError: 'tuple' object does not support item assignment
```

Dictionaries

Mapping from keys to values:

```
Yade [30]: czde={'jedna':'ein','dva':'zwei','tri':'drei'}

Yade [31]: de={1:'ein',2:'zwei',3:'drei'}; cz={1:'jedna',2:'dva',3:'tri'}

Yade [32]: czde['jedna']          ## access values
Out[32]: 'ein'

Yade [33]: de[1], cz[2]
Out[33]: ('ein', 'dva')
```

Functions, conditionals

```
Yade [34]: 4==5
Out[34]: False

Yade [35]: a=3.1

Yade [36]: if a<pi: b=0           # conditional statement
   ....:  else: b=1
   ....:
  File "<ipython-input-37-bebe87cdf86d>", line 1
    else: b=1
       ^
SyntaxError: invalid syntax


Yade [38]: c=0 if a<1 else 1      # conditional expression

Yade [39]: def square(x): return x**2    # define a new function
   ....:

Yade [40]: square(2)                     # and call that function
Out[40]: 4
```

Exercises

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

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

3.2.3 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"):

```
Yade [42]: Sphere           # try also Sphere?
Out[42]: yade.wrapper.Sphere

Yade [43]: s=Sphere()       # create a Sphere, without specifying any attributes

Yade [44]: s.radius         # 'nan' is a special value meaning "not a number" (i.e. not defined)
Out[44]: nan

Yade [45]: s.radius=2       # set radius of an existing object

Yade [46]: s.radius
Out[46]: 2.0

Yade [47]: ss=Sphere(radius=3)   # create Sphere, giving radius directly

Yade [48]: s.radius, ss.radius     # also try typing s.<tab> to see defined attributes
Out[48]: (2.0, 3.0)
```

Particles

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

Singles

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

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

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

```
Yade [50]: s=utils.sphere((0,0,0),radius=1)    # create sphere particle centered at (0,0,0) with radius=1

Yade [51]: s.shape                      # s.shape describes the geometry of the particle
Out[51]: <Sphere instance at 0x32e01e0>

Yade [52]: s.shape.radius               # we already know the Sphere class
Out[52]: 1.0
```

```
Yade [53]: s.state.mass, s.state.inertia # inertia is computed from density and geometry
Out[53]:
(4188.790204786391,
 Vector3(1675.5160819145563,1675.5160819145563,1675.5160819145563))

Yade [54]: s.state.pos                   # position is the one we prescribed
Out[54]: Vector3(0,0,0)

Yade [55]: s2=utils.sphere((-2,0,0),radius=1,fixed=True)     # explanation below
```

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

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

```
Yade [56]: O.bodies.append(s)            # adds particle s to the simulation; returns id of the particle(s) add
Out[56]: 13
```

Packs

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

```
Yade [57]: from yade import pack

Yade [58]: sp=pack.SpherePack()                  # create an empty cloud; SpherePack contains only geometrical

Yade [59]: sp.makeCloud((1,1,1),(2,2,2),rMean=.2) # put spheres with defined radius inside box given by corners
Out[59]: 7

Yade [60]: for c,r in sp: print c,r              # print center and radius of all particles (SpherePack is a s
   ....:
Vector3(1.6954145940909298,1.7578737141172156,1.722055098993755) 0.2
Vector3(1.4499560608062483,1.429019046508725,1.778402361533048) 0.2
Vector3(1.2810537813054899,1.7470890621161916,1.3361185998060914) 0.2
Vector3(1.3365789895286588,1.2044139543589334,1.266004247372974) 0.2
Vector3(1.7437108145502496,1.4647304202101477,1.2021214999278276) 0.2
Vector3(1.791152070268199,1.3015841148808451,1.566820616523568) 0.2
Vector3(1.2464497237898873,1.7746295043031028,1.740802785326543) 0.2

Yade [61]: sp.toSimulation()                     # create particles and add them to the simulation
Out[61]: [14, 15, 16, 17, 18, 19, 20]
```

Boundaries

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

```
Yade [62]: O.bodies.append(utils.wall(-1,axis=2))
Out[62]: 21
```

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

Look inside

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

```
Yade [63]: len(O.bodies)
Out[63]: 22

Yade [64]: O.bodies[1].shape.radius   # radius of body #1 (will give error if not sphere, since only spheres ha
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/usr/lib/x86_64-linux-gnu/yadedaily/py/yade/__init__.pyc in <module>()
----> 1 O.bodies[1].shape.radius   # radius of body #1 (will give error if not sphere, since only spheres have

AttributeError: 'NoneType' object has no attribute 'shape'

Yade [65]: O.bodies[2].state.pos      # position of body #2
Out[65]: Vector3(0,0,0)
```

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?

   ```
   Yade [66]: O.bodies.append([utils.sphere((2*i,0,0),1) for i in range(1,20)])
   Out[66]: [22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40]
   ```

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:

```
Yade [68]: O.engines=[                    # newlines and indentations are not important until the brace is close
   ....:     ForceResetter(),
   ....:     InsertionSortCollider([Bo1_Sphere_Aabb(),Bo1_Wall_Aabb()]),
   ....:     InteractionLoop(           # dtto for the parenthesis here
   ....:         [Ig2_Sphere_Sphere_L3Geom(),Ig2_Wall_Sphere_L3Geom()],
   ....:         [Ip2_FrictMat_FrictMat_FrictPhys()],
   ....:         [Law2_L3Geom_FrictPhys_ElPerfPl()]
```

```
....:     ),
....:     NewtonIntegrator(damping=.2,label='newton')      # define a name under which we can access this en
....: ]
....:

Yade [69]: O.engines
Out[69]:
[<ForceResetter instance at 0x368fd20>,
 <InsertionSortCollider instance at 0x3325390>,
 <InteractionLoop instance at 0x34a06a0>,
 <NewtonIntegrator instance at 0x313f440>]

Yade [70]: O.engines[-1]==newton    # is it the same object?
Out[70]: True

Yade [71]: newton.damping
Out[71]: 0.2
```

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

```
O.engines=[                    # newlines and indentations are not important until the brace is closed
        ForceResetter(),
        InsertionSortCollider([Bo1_Sphere_Aabb(),Bo1_Wall_Aabb()]),
        InteractionLoop(           # dtto for the parenthesis here
                [Ig2_Sphere_Sphere_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`:

```
Yade [72]: O.dt=0.01

Yade [73]: O.time
Out[73]: 0.0

Yade [74]: O.step()

Yade [75]: O.time
Out[75]: 0.01
```

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

```
O.saveTmp()
```

Exercises

1. Define *engines* as in the above example, run the *Inspector* and click through the engines to see their sequence.
2. Write a simple script which will
   (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.

3.3 Data mining

3.3.1 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.

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

Yade [3]: newton.damping=.4

Yade [4]: O.engines[0].damping   # O.engines[0] and newton are the same objects
Out[4]: 0.4
```

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]`.

3.3.2 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.

3.3.3 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.

3.4 Towards geomechanics

See also:

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

3.4.1 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.

3.4.2 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).

```
Yade [2]: sp=pack.SpherePack()

Yade [3]: sp.makeCloud((1,1,1),(2,2,2),rMean=.2,periodic=True)
Out[3]: 9

Yade [4]: sp.toSimulation()             # implicitly sets O.periodic=True, and O.cell.refSize to the packing pe
Out[4]: [4, 5, 6, 7, 8, 9, 10, 11, 12]

Yade [5]: O.engines+=[PeriTriaxController(goal=(-1e4,-1e4,-.1),stressMask=0b011,maxUnbalanced=.2,doneHook='func
```

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()
```

3.5 Advanced & more

3.5.1 Particle size distribution

See *Periodic triaxial test*

3.5.2 Clumps

Clump; see *Periodic triaxial test*

3.5.3 Testing laws

LawTester, scripts/test/law-test.py

3.5.4 New law

3.5.5 Visualization

See the example *3d postprocessing*

- VTKRecorder & Paraview
- qt.SnapshotEngine

3.6 Examples

3.6.1 Bouncing sphere

```
# 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()
```

3.6.2 Gravity deposition

```
# 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()
```

3.6.3 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
```

3.6.4 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()
```

3.6.5 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
```

3.6.6 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

O.periodic=True
sp=pack.SpherePack()
if 0:
```

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

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

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

```
    triax.doneHook='triaxFinished()'
    # do not wait for stabilization before calling triaxFinished
    triax.maxUnbalanced=10

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

Chapter 4

User's manual

4.1 Scene construction

4.1.1 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 sphere, facet and wall are used to create them. Using these functions also ensures better future compatibility, if internals of Body change in some way. These functions receive geometry of the particle and several other characteristics. See their documentation for details. If the same Material is used for several (or many) bodies, it can be shared by adding it in `O.materials`, as explained below.

Defining materials

The `O.materials` object (instance of Omega.materials) holds defined shared materials for bodies. It only supports addition, and will typically hold only a few instance (though there is no limit).

`label` given to each material is optional, but can be passed to sphere and other functions for constructing body. The value returned by `O.materials.append` is an `id` of the material, which can be also passed to sphere – it is a little bit faster than using label, though not noticeable for small number of particles and perhaps less convenient.

If no Material is specified when calling sphere, the *last* defined material is used; that is a convenient default. If no material is defined yet (hence there is no last material), a default material will be created: FrictMat(density=2e3,young=30e9,poisson=.3,frictionAngle=.5236). This should not happen for serious simulations, but is handy in simple scripts, where exact material properties are more or less irrelevant.

```
Yade [2]: len(O.materials)
 Out[2]: 0

Yade [3]: idConcrete=O.materials.append(FrictMat(young=30e9,poisson=.2,frictionAngle=.6,label="concrete"))

Yade [4]: O.materials[idConcrete]
 Out[4]: <FrictMat instance at 0x33de130>

# uses the last defined material
Yade [5]: O.bodies.append(sphere(center=(0,0,0),radius=1))
 Out[5]: 0
```

```
# material given by id
Yade [6]: O.bodies.append(sphere((0,0,2),1,material=idConcrete))
 Out[6]: 1

# material given by label
Yade [7]: O.bodies.append(sphere((0,2,0),1,material="concrete"))
 Out[7]: 2

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

Yade [9]: len(O.materials)
 Out[9]: 2

# implicitly uses "steel" material, as it is the last one now
Yade [10]: O.bodies.append(facet([(1,0,0),(0,1,0),(-1,-1,0)]))
Out[10]: 3
```

Adding multiple particles

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

```
Yade [2]: O.bodies.append(sphere((0,10,0),1))
 Out[2]: 0

Yade [3]: O.bodies.append(sphere((0,0,2),1))
 Out[3]: 1

# this is the same, but in one function call
Yade [4]: O.bodies.append([
   ...:    sphere((0,0,0),1),
   ...:    sphere((0,0,2),1)
   ...:  ])
   ...:
 Out[4]: [2, 3]
```

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

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

As those functions use sphere and facet internally, they accept additional 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). This we call a clump. A clump is internally represented by a special body, referenced by clumpId of its members (see also isClump, isClumpMember and isStandalone). Like every body a clump has a position, which is the (mass) balance point between all members. A clump body itself has no interactions with other bodies. Interactions between clumps is represented by interactions between clump members. There are no interactions between clump members of the same clump.

YADE supports different ways of creating clumps:

- Create clumps and spheres (clump members) directly with one command:

The function appendClumped() is designed for this task. For instance, we might add 2 spheres tied together:

```
Yade [2]: O.bodies.appendClumped([
   ...:     sphere([0,0,0],1),
   ...:     sphere([0,0,2],1)
   ...:  ])
   ...:
 Out[2]: (2, [0, 1])

Yade [3]: len(O.bodies)
 Out[3]: 3

Yade [4]: O.bodies[1].isClumpMember, O.bodies[2].clumpId
 Out[4]: (True, 2)

Yade [5]: O.bodies[2].isClump, O.bodies[2].clumpId
 Out[5]: (True, 2)
```

-> appendClumped() returns a tuple of ids `(clumpId,[memberId1,memberId2,...])`

- Use existing spheres and clump them together:

For this case the function clump() can be used. One way to do this is to create a list of bodies, that should be clumped before using the clump() command:

```
Yade [2]: bodyList = []

Yade [3]: for ii in range(0,5):
   ...:     bodyList.append(O.bodies.append(sphere([ii,0,1],.5)))#create a "chain" of 5 spheres
   ...:

Yade [4]: print bodyList
[0, 1, 2, 3, 4]

Yade [5]: idClump=O.bodies.clump(bodyList)
```

-> clump() returns `clumpId`

- Another option is to replace standalone spheres from a given packing (see SpherePack and makeCloud) by clumps using clump templates.

This is done by a function called replaceByClumps(). This function takes a list of clumpTemplates() and a list of amounts and replaces spheres by clumps. The volume of a new clump will be the same as the volume of the sphere, that was replaced (clump volume/mass/inertia is accounting for overlaps assuming that there are only pair overlaps).

-> replaceByClumps() returns a list of tuples: `[(clumpId1,[memberId1,memberId2,...]),(clumpId2,[memberId1,mem`

It is also possible to add bodies to a clump and release bodies from a clump. Also you can erase the clump (clump members will get standalone spheres).

Additionally YADE supports to achieve the roundness of a clump or roundness coefficient of a packing. Parts of the packing can be excluded from roundness measurement via exclude list.

```
Yade [2]: bodyList = []

Yade [3]: for ii in range(1,5):
   ...:     bodyList.append(O.bodies.append(sphere([ii,ii,ii],.5)))
   ...:

Yade [4]: O.bodies.clump(bodyList)
 Out[4]: 4

Yade [5]: RC=O.bodies.getRoundness()
```

```
Yade [6]: print RC
0.256191414232
```

-> getRoundness() returns roundness coefficient RC of a packing or a part of the packing

Note: Have a look at examples/clumps/ folder. There you will find some examples, that show usage of different functions for clumps.

4.1.2 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)
```

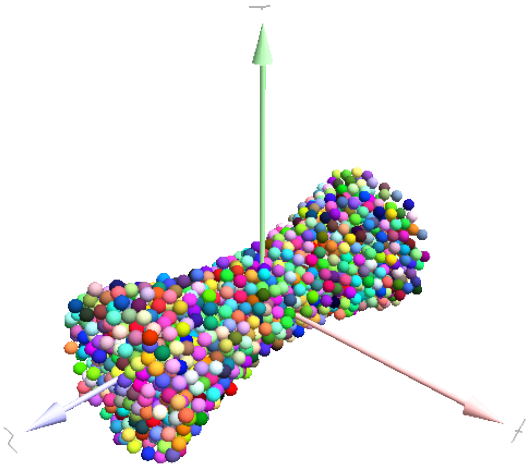


Figure 4.1: Specimen constructed with the pack.inHyperboloid predicate, packed with pack.randomDensePack.

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))
```

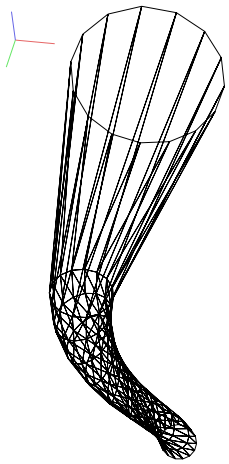


Figure 4.2: Triangulated funnel, constructed with the examples/funnel.py script.

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 examples/gts-horse/gts-horse.py (img. img-horse) shows both possibilities; first, a GTS surface is imported:

```
import gts
surf=gts.read(open('horse.coarse.gts'))
```

That surface object is used as predicate for packing:

```
pred=pack.inGtsSurface(surf)
O.bodies.append(pack.regularHexa(pred,radius=radius,gap=radius/4.))
```

and then, after being translated, as base for triangulated surface in the simulation itself:

```
surf.translate(0,0,-(aabb[1][2]-aabb[0][2]))
O.bodies.append(pack.gtsSurface2Facets(surf,wire=True))
```

Boolean operations on predicates

Boolean operations on pair of predicates (noted `A` and `B`) are defined:

- intersection `A & B` (conjunction): point must be in both predicates involved.
- union `A | B` (disjunction): point must be in the first or in the second predicate.
- difference `A - B` (conjunction with second predicate negated): the point must be in the first predicate and not in the second one.
- symmetric difference `A ^ B` (exclusive disjunction): point must be in exactly one of the two predicates.

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

```
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)
```

Figure 4.3: 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.

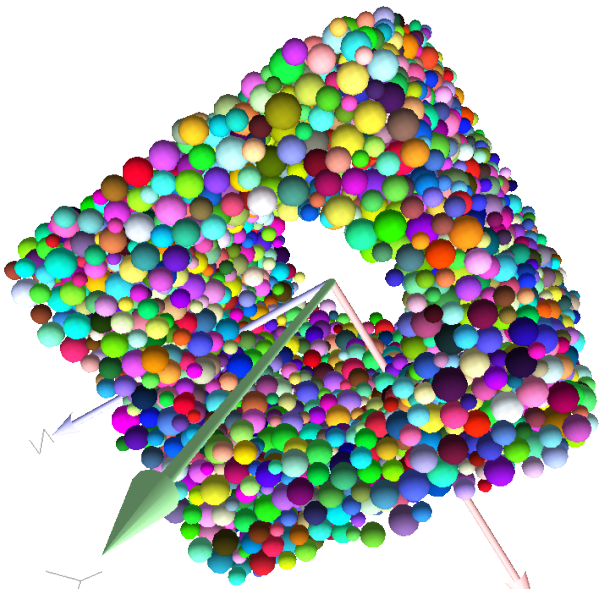


Figure 4.4: Box with cylinder removed from inside, using difference of these two predicates.

Packing algorithms

Algorithms presented below operate on geometric spheres, defined by their center and radius. With a few exception documented below, the procedure is as follows:

1. Sphere positions and radii are computed (some functions use volume predicate for this, some do not)
2. sphere is called for each position and radius computed; it receives extra keyword arguments of the packing function (i.e. arguments that the packing function doesn't specify in its definition; they are noted `**kw`). Each sphere call creates actual Body objects with Sphere shape. List of Body objects is returned.
3. List returned from the packing function can be added to simulation using `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 sphere, where they are also documented. `spheres` is now list of Body objects, which we add to the simulation:

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

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

```
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(sphere(c,r))
```

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

```
O.bodies.append([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.

GenGeo is library (python module) for packing generation developed with ESyS-Particle. It creates packing by random insertion of spheres with given radius range. Inserted spheres touch each other exactly and, more importantly, they also touch the boundary, if in its neighbourhood. Boundary is represented as special object of the GenGeo library (Sphere, cylinder, box, convex polyhedron, ...). Therefore, GenGeo cannot be used with volume represented by yade predicates as explained above.

Packings generated by this module can be imported directly via ymport.gengeo, or from saved file via ymport.gengeoFile. There is an example script examples/test/genCylLSM.py. Full documentation for GenGeo can be found at ESyS documentation website.

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.

4.1.3 Triangulated surfaces

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

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

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

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

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

Import

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

ymport.gts text file in native GTS format.

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

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

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

Parametric construction

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

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

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

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

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

4.1.4 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 (Ig2_Sphere_Sphere_ScGeom.interactionDetectionFactor).

   Note that only Sphere + Sphere interactions are supported; there is no parameter analogous to distFactor in Ig2_Facet_Sphere_ScGeom. This is on purpose, since the interaction radius is meaningful in bulk material represented by sphere packing, whereas facets usually represent boundary conditions which should be exempt from this dense interaction network.

3. Run one single step of the simulation so that the initial network is created.
4. Reset interaction radius in both `Bo1` and `Ig2` functors to their default value again.
5. Continue the simulation; interactions that are already established will not be deleted (the `Law2` functor in 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_ScGeom(interactionDetectionFactor=intRadius,label='ig2ss'),
         Ig2_Facet_Sphere_ScGeom(),
      ],
      [Ip2_CpmMat_CpmMat_CpmPhys()],
      [Law2_ScGeom_CpmPhys_Cpm(epsSoft=0)], # deactivated
   ),
   NewtonIntegrator(damping=damping,label='damper'),
]

# run one single step
O.step()

# reset interaction radius to the default value
bo1s.aabbEnlargeFactor=1.0
ig2ss.interactionDetectionFactor=1.0

# now continue simulation
O.run()
```

Individual interactions on demand

It is possible to create an interaction between a pair of particles independently of collision detection using createInteraction. This function looks for and uses matching `Ig2` and `Ip2` functors. Interaction will be created regardless of distance between given particles (by passing a special parameter to the `Ig2` functor to force creation of the interaction even without any geometrical contact). Appropriate constitutive law should be used to avoid deletion of the interaction at the next simulation step.

```
Yade [2]: O.materials.append(FrictMat(young=3e10,poisson=.2,density=1000))
 Out[2]: 0

Yade [3]: O.bodies.append([
   ...:     sphere([0,0,0],1),
   ...:     sphere([0,0,1000],1)
   ...:  ])
```

```
   ...:
 Out[3]: [0, 1]

# only add InteractionLoop, no other engines are needed now
Yade [4]: O.engines=[
   ...:     InteractionLoop(
   ...:         [Ig2_Sphere_Sphere_ScGeom(),],
   ...:         [Ip2_FrictMat_FrictMat_FrictPhys()],
   ...:         [] # not needed now
   ...:     )
   ...:  ]
   ...:

Yade [5]: i=createInteraction(0,1)

# created by functors in InteractionLoop
Yade [6]: i.geom, i.phys
 Out[6]: (<ScGeom instance at 0x3074320>, <FrictPhys instance at 0x362bf50>)
```

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 createInteraction function accept list of pairs id's as well.

4.1.5 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:]
```

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

Functors choice

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

Bo1 functors

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

At this moment (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_ScGeom()],[],[])
   ```

   will handle collisions for Sphere + Sphere, but not for Facet + Sphere – if that is desired, an additional functor must be used:

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

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

2. IGeom type accepted by the `Law2` functor (below); it is the first part of functor's name after `Law2` (for instance, Law2_ScGeom_CpmPhys_Cpm accepts ScGeom).

Ip2 functors

`Ip2` functors (deriving from IPhysFunctor) must be chosen depending on

1. Material combinations within the simulation. In most cases, `Ip2` functors handle 2 instances of the same Material class (such as Ip2_FrictMat_FrictMat_FrictPhys for 2 bodies with FrictMat)
2. IPhys accepted by the constitutive law (`Law2` functor), which is the second part of the `Law2` functor's name (e.g. Law2_ScGeom_FrictPhys_CundallStrack accepts FrictPhys)

Note: Unlike with `Bo1` and `Ig2` functors, unhandled combination of Materials is an error condition signaled by an exception.

Law2 functor(s)

`Law2` functor was the ultimate criterion for the choice of `Ig2` and `Ip2` functors; there are no restrictions on its choice in itself, as it only applies forces without creating new objects.

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

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

Warning: Many `Law2` exist in Yade, and new ones can appear at any time. In some cases different functors are only different implementations of the same contact law (e.g. Law2_ScGeom_FrictPhys_CundallStrack and Law2_L3Geom_FrictPhys_ElPerfPl). Also, sometimes, the peculiarity of one functor may be reproduced as a special case of a more general one. Therefore, for a given constitutive behavior, the user may have the choice between different functors. It is strongly recommended to favor the most used and most validated implementation when facing such choice. A list of available functors classified from mature to unmaintained is updated here to guide this choice.

Examples

Let us give several 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_ScGeom_CpmPhys_Cpm constitutive law.

- We use spheres and facets in the simulation, which selects `Ig2` functors accepting those types and producing ScGeom: Ig2_Sphere_Sphere_ScGeom and Ig2_Facet_Sphere_ScGeom.
- We have to use Material which can be used for creating CpmPhys. We find that CpmPhys is only created by Ip2_CpmMat_CpmMat_CpmPhys, which determines the choice of CpmMat for all particles.

Therefore, we will use:

```
InteractionLoop(
   [Ig2_Sphere_Sphere_ScGeom(),Ig2_Facet_Sphere_ScGeom()],
   [Ip2_CpmMat_CpmMat_CpmPhys()],
   [Law2_ScGeom_CpmPhys_Cpm()]
)
```

4.1.6 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 accelerated by NewtonIntegrator; it is mandatory to make it false for bodies with zero mass, where applying non-zero force would result in infinite displacement.

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

  Making sphere non-dynamic is achieved simply by:

  ```
  b = sphere([x,y,z],radius,dynamic=False)
  b.dynamic=True #revert the previous
  ```

- State.blockedDOFs permits selective blocking of any of 6 degrees of freedom in global space. For instance, a sphere can be made to move only in the xy plane by saying:

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

  Yade [3]: O.bodies[0].state.blockedDOFs='zXY'
  ```

  In contrast to Body.dynamic, blockedDOFs will only block forces (and acceleration) in selected directions. Actually, `b.dynamic=False` is nearly only a shorthand for `b.state.blockedDOFs=='xyzXYZ'` . A subtle difference is that the former does reset the velocity components automaticaly, while the latest does not. If you prescribed linear or angular velocity, they will be applied regardless of blockedDOFs. It also implies that if the velocity is not zero when degrees of freedom are blocked via blockedDOFs assignements, the body will keep moving at the velocity it has at the time of blocking. The differences are shown below:

  ```
  Yade [2]: b1 = sphere([0,0,0],1,dynamic=True)

  Yade [3]: b1.state.blockedDOFs
  ```

```
Out[3]: ''

Yade [4]: b1.state.vel = Vector3(1,0,0) #we want it to move...

Yade [5]: b1.dynamic = False #... at a constant velocity

Yade [6]: print b1.state.blockedDOFs, b1.state.vel
xyzXYZ Vector3(0,0,0)

Yade [7]: # oops, velocity has been reset when setting dynamic=False

Yade [8]: b1.state.vel = (1,0,0) # we can still assign it now

Yade [9]: print b1.state.blockedDOFs, b1.state.vel
xyzXYZ Vector3(1,0,0)

Yade [10]: b2 = sphere([0,0,0],1,dynamic=True) #another try

Yade [11]: b2.state.vel = (1,0,0)

Yade [12]: b2.state.blockedDOFs = "xyzXYZ" #this time we assign blockedDOFs directly, velocity is unchange

Yade [13]: print b2.state.blockedDOFs, b2.state.vel
xyzXYZ Vector3(1,0,0)
```

It might be desirable to constrain motion of some particles constructed from a generated sphere packing, following some condition, such as being at the bottom of a specimen; this can be done by looping over all bodies with a conditional:

```
for b in O.bodies:
   # block all particles with z coord below .5:
   if b.state.pos[2]<.5: b.dynamic=False
```

Arbitrary spatial predicates introduced above can be expoited here as well:

```
from yade import pack
pred=pack.inAlignedBox(lowerCorner,upperCorner)
for b in O.bodies:
   if 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
```

Imposing motion and forces

Imposed velocity

If a degree of freedom is blocked and a velocity is assigned along that direction (translational or rotational velocity), then the body will move at constant velocity. This is the simpler and recommended method to impose the motion of a body. This, for instance, will result in a constant velocity along x (it can still be freely accelerated along y and z):

```
O.bodies.append(sphere((0,0,0),1))
O.bodies[0].state.blockedDOFs='x'
O.bodies[0].state.vel=(10,0,0)
```

Conversely, modifying the position directly is likely to break Yade's algorithms, especially those related to collision detection and contact laws, as they are based on bodies velocities. Therefore, unless you really know what you are doing, don't do that for imposing a motion:

```
O.bodies.append(sphere((0,0,0),1))
O.bodies[0].state.blockedDOFs='x'
O.bodies[0].state.pos=10*O.dt #REALLY BAD! Don't assign position
```

Imposed force

Applying a force or a torque on a body is done via functions of the ForceContainer. It is as simple as this:

```
O.forces.addF(0,(1,0,0)) #applies for one step
```

By default, the force applies for one time step only, and is resetted at the beginning of each step. For this reason, imposing a force at the begining of one step will have no effect at all, since it will be immediatly resetted. The only way is to place a PyRunner inside the simulation loop.

Applying the force permanently is possible with an optional argument (in this case it does not matter if the command comes at the begining of the time step):

```
O.forces.addF(0,(1,0,0),permanent=True) #applies permanently
```

The force will persist across iterations, until it is overwritten by another call to `O.forces.addF(id,f,True)` or erased by `O.forces.reset(resetAll=True)`. The permanent force on a body can be checked with `O.forces.permF(id)`.

Boundary controllers

Engines deriving from BoundaryController impose boundary conditions during simulation, either directly, or by influencing several bodies. You are referred to their individual documentation for details, though you might find interesting in particular

- UniaxialStrainer for applying strain along one axis at constant rate; useful for plotting strain-stress diagrams for uniaxial loading case. See examples/concrete/uniax.py for an example.
- TriaxialStressController which applies prescribed stress/strain along 3 perpendicular axes on cuboid-shaped packing using 6 walls (Box objects) (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 (GravityEngine is deprecated, use NewtonIntegrator.gravity instead!).

Partial engines

Engines deriving from PartialEngine define the ids attribute determining bodies which will be affected. Several of them warrant explicit mention here:

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

The real value of partial engines is if you need to prescribe complex types of force or displacement fields. For moving a body at constant velocity or for imposing a single force, the methods explained in Imposing motion and forces are much simpler. There are several interpolating engines (InterpolatingDirectedForceEngine for applying force with varying magnitude, InterpolatingHelixEngine for applying spiral displacement with varying angular velocity and possibly others); writing a new interpolating engine is rather simple using examples of those that already exist.

4.1.7 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:

```
Yade [2]: dict(O.tags) # convert to real dictionary
 Out[2]:
{'author': '~(bchareyre@dt-rv020)',
 'd.id': '20151119T215521p31818',
 'id': '20151119T215521p31818',
 'id.d': '20151119T215521p31818',
 'isoTime': '20151119T215521'}
```

author Real name, username and machine as obtained from your system at simulation creation

id Unique identifier of this Yade instance (or of the instance which created a loaded simulation). It is composed of date, time and process number. Useful if you run simulations in parallel and want to avoid overwriting each other's outputs; embed `O.tags['id']` in output filenames (either as directory name, or as part of the file's name itself) to avoid it. This is explained in *Separating output files from jobs* in detail.

isoTime Time when simulation was created (with second resolution).

d.id, id.d Simulation description and id joined by period (and vice-versa). Description is used in batch jobs; in non-batch jobs, these tags are identical to id.

You can add your own tags by simply assigning value, with the restriction that the left-hand side object must be a string and must not contain `=`.

```
Yade [1]: O.tags['anythingThat I lik3']='whatever'

Yade [2]: O.tags['anythingThat I lik3']
 Out[2]: 'whatever'
```

Saving python variables

Python variable lifetime is limited; in particular, if you save simulation, variables will be lost after reloading. Yade provides limited support for data persistence for this reason (internally, it uses special values of `O.tags`). The functions in question are saveVars and loadVars.

saveVars takes dictionary (variable names and their values) and a *mark* (identification string for the variable set); it saves the dictionary inside the simulation. These variables can be re-created (after the simulation was loaded from a XML file, for instance) in the `yade.params.`*mark* namespace by calling loadVars with the same identification *mark*:

```
Yade [1]: a=45; b=pi/3

Yade [2]: saveVars('ab',a=a,b=b)

# save simulation (we could save to disk just as well)
Yade [2]: O.saveTmp()

Yade [4]: O.loadTmp()

Yade [5]: loadVars('ab')

Yade [6]: yade.params.ab.a
 Out[6]: 45

# import like this
Yade [7]: from yade.params import ab

Yade [8]: ab.a, ab.b
 Out[8]: (45, 1.0471975511965976)

# also possible
Yade [9]: from yade.params import *

Yade [10]: ab.a, ab.b
Out[10]: (45, 1.0471975511965976)
```

Enumeration of variables can be tedious if they are many; creating local scope (which is a function definition in Python, for instance) can help:

```
def setGeomVars():
        radius=a*4
        thickness=22
        p_t=4/3*pi
        dim=Vector3(1.23,2.2,3)
        #
        # define as much as you want here
        # it all appears in locals() (and nothing else does)
        #
        saveVars('geom',loadNow=True,**locals())

setGeomVars()
from yade.params.geom import *
# use the variables now
```

Note: Only types that can be pickled can be passed to saveVars.

4.2 Controlling simulation

4.2.1 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 kineticEnergy) every 5 seconds, the following engine will be put to `O.engines`:

```
PyRunner(command="print 'kinetic energy',kineticEnergy()",realPeriod=5)
```

For running more complex commands, it is convenient to define an external function and only call it from within the engine. Since the `command` is run in the script's namespace, functions defined within scripts can be called. Let us print information on interaction between bodies 0 and 1 periodically:

```
def intrInfo(id1,id2):
        try:
                i=O.interactions[id1,id2]
                # assuming it is a CpmPhys instance
                print id1,id2,i.phys.sigmaN
        except:
                # in case the interaction doesn't exist (yet?)
                print "No interaction between",id1,id2
O.engines=[...,
        PyRunner(command="intrInfo(0,1)",realPeriod=5)
]
```

More useful examples will be given below.

The plot module provides simple interface and storage for tracking various data. Although originally conceived for plotting only, it is widely used for tracking variables in general.

The data are in plot.data dictionary, which maps variable names to list of their values; the plot.addData function is used to add them.

```
Yade [2]: from yade import plot

Yade [3]: plot.data
 Out[3]: {}

Yade [4]: plot.addData(sigma=12,eps=1e-4)

# not adding sigma will add a NaN automatically
# this assures all variables have the same number of records
Yade [5]: plot.addData(eps=1e-3)

# adds NaNs to already existing sigma and eps columns
Yade [6]: plot.addData(force=1e3)

Yade [7]: plot.data
 Out[7]:
{'eps': [0.0001, 0.001, nan],
 'force': [nan, nan, 1000.0],
 'sigma': [12, nan, nan]}

# retrieve only one column
Yade [8]: plot.data['eps']
 Out[8]: [0.0001, 0.001, nan]

# get maximum eps
Yade [9]: max(plot.data['eps'])
 Out[9]: 0.001
```

New record is added to all columns at every time plot.addData is called; this assures that lines in different columns always match. The special value `nan` or `NaN` (Not a Number) is inserted to mark the record invalid.

Note: It is not possible to have two columns with the same name, since data are stored as a dictionary.

To record data periodically, use PyRunner. This will record the z coordinate and velocity of body #1, iteration number and simulation time (every 20 iterations):

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

from yade import plot
def myAddData():
        b=O.bodies[1]
        plot.addData(z1=b.state.pos[2], v1=b.state.vel.norm(), i=O.iter, t=O.time)
```

Note: Arbitrary string can be used as column label for plot.data. If it cannot be used as keyword name for plot.addData (since it is a python keyword (`for`), or has spaces inside (`my funny column`), you can pass dictionary to plot.addData instead:

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

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

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

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

In that case, naturally, the labeled object must define attributes which are used (UniaxialStrainer.strain and UniaxialStrainer.avgStress in this case).

Plotting variables

Above, we explained how to track variables by storing them using plot.addData. These data can be readily used for plotting. Yade provides a simple, quick to use, plotting in the plot module. Naturally, since direct access to underlying data is possible via plot.data, these data can be processed in any 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')}
```

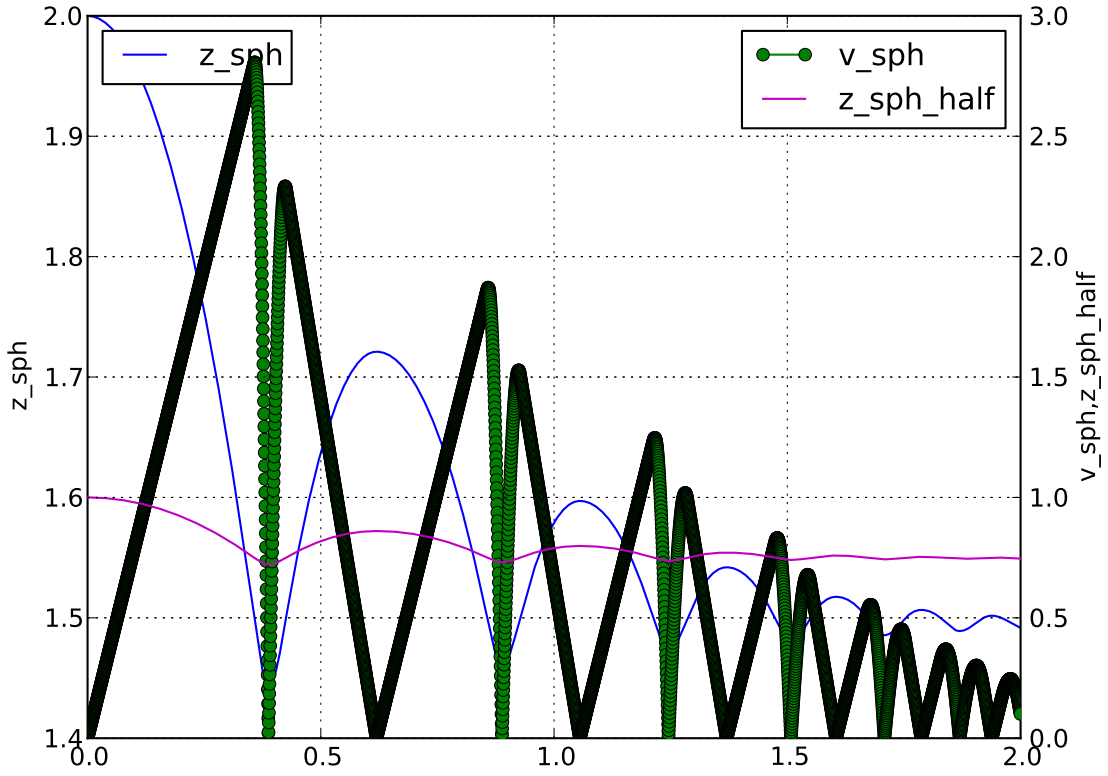


Figure 4.5: Figure generated by examples/simple-scene/simple-scene-plot.py.

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:

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

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:

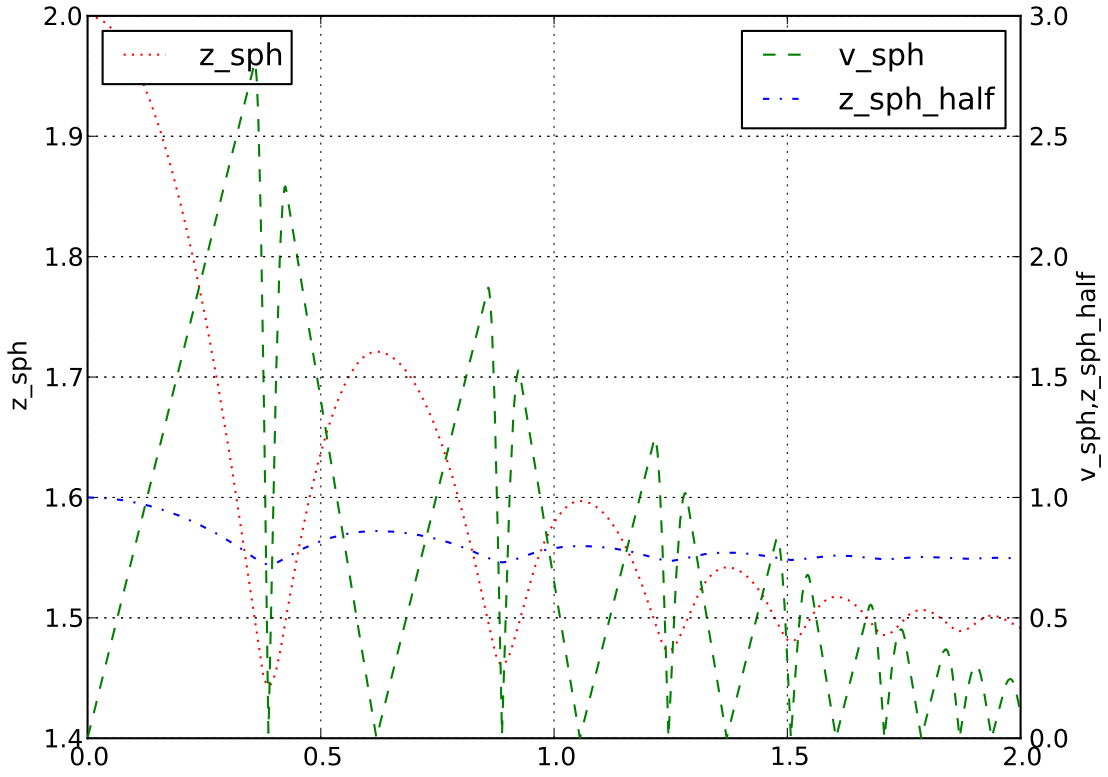


Figure 4.6: Figure generated by changing parameters to plot.plots as above.

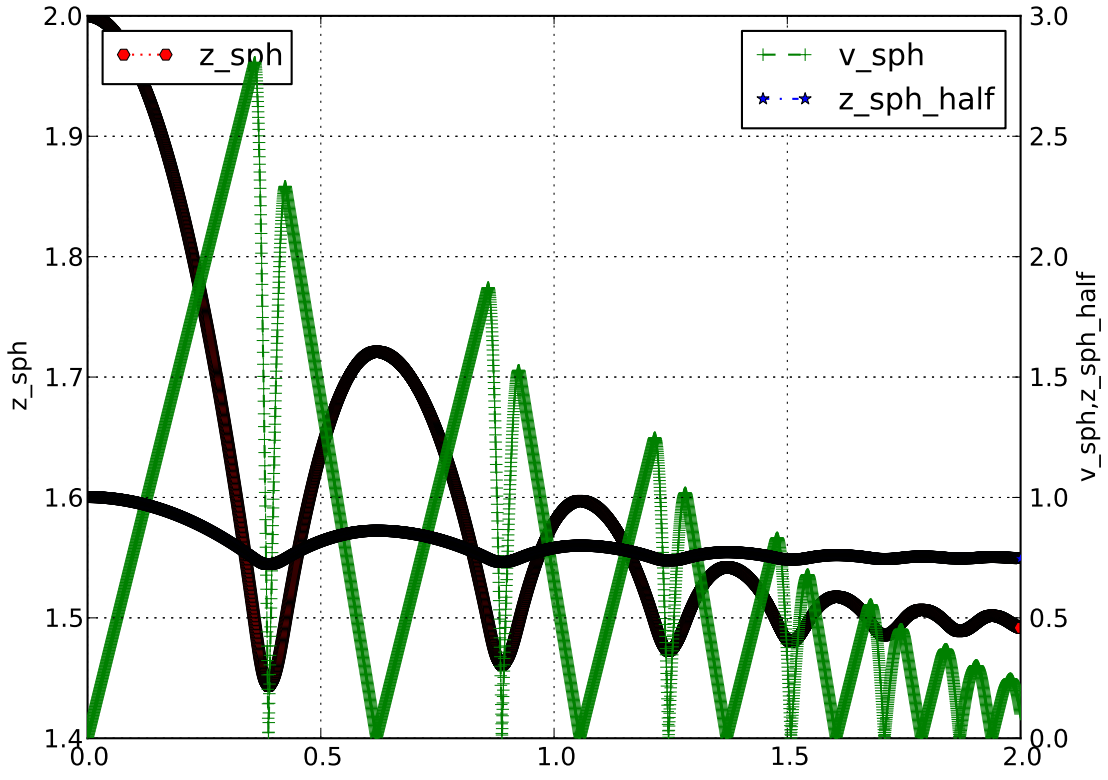


Figure 4.7: Figure generated by changing parameters to plot.plots as above.

```
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':'$z_{sph}/2$'}
```

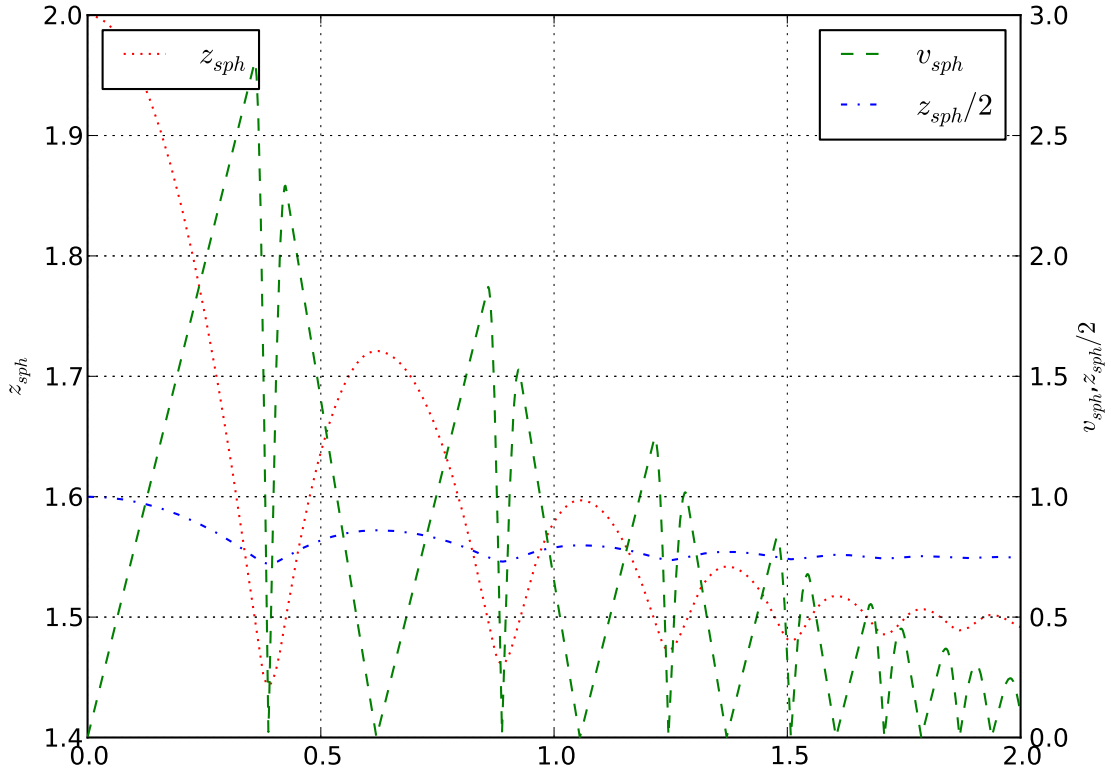


Figure 4.8: Figure generated by examples/simple-scene/simple-scene-plot.py, with TeX labels.

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

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

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

Multiple figures

Since plot.plots is a dictionary, multiple entries with the same key (x-axis variable) would not be possible, since they overwrite each other:

```
Yade [1]: plot.plots={
   ...:     'i':('t',),
   ...:     'i':('z1','v1')
   ...:  }
   ...:

Yade [2]: plot.plots
 Out[2]: {'i': ('z1', 'v1')}
```

You can, however, distinguish them by prepending/appending space to the x-axis variable, which will be removed automatically when looking for the variable in plot.data – both x-axes will use the `i` column:

```
Yade [1]: plot.plots={
   ...:     'i':('t',),
   ...:     'i ':('z1','v1') # note the space in 'i '
   ...:  }
   ...:

Yade [2]: plot.plots
 Out[2]: {'i': ('t',), 'i ': ('z1', 'v1')}
```

Split y1 y2 axes

To avoid big range differences on the `y` axis, it is possible to have left and right `y` axes separate (like `axes x1y2` in gnuplot). This is achieved by inserting `None` to the plot specifier; variables coming before will be plot normally (on the left y-axis), while those after will appear on the right:

```
plot.plots={'i':('z1',None,'v1')}
```

Exporting

Plots can be exported to external files for later post-processing via that plot.saveGnuplot function. Note that all data you added via plot.addData is saved - even data that you don't plot live during simulation. By editing the generated .gnuplot file you can plot any of the added Data afterwards.

- Data file is saved (compressed using bzip2) separately from the gnuplot file, so any other programs can be used to process them. In particular, the `numpy.genfromtxt` (documented here) can be useful to import those data back to python; the decompression happens automatically.
- The gnuplot file can be run through gnuplot to produce the figure; see plot.saveGnuplot documentation for details.

4.2.2 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 unbalancedForce can provide a useful metrics (see its documentation for details); for a desired value of `1e-2` or less, for instance, we can use:

```
def checkUnbalanced():
        if unbalancedForce<1e-2: O.pause()

O.engines=O.engines+[PyRunner(command="checkUnbalanced()",iterPeriod=100)]

# this would work as well, without the function defined apart:
#   PyRunner(command="if unablancedForce<1e-2: O.pause()",iterPeriod=100)

O.run(); O.wait()
# will continue after O.pause() will have been called
```

Arbitrary functions can be periodically checked, and they can also use history of variables tracked via plot.addData. For example, this is a simplified version of damage control in examples/concrete/uniax.py; it stops when current stress is lower than half of the peak stress:

```
O.engines=[...,
        UniaxialStrainer=(...,label='strainer'),
        PyRunner(command='myAddData()',iterPeriod=100),
        PyRunner(command='stopIfDamaged()',iterPeriod=100)
]

def myAddData():
        plot.addData(t=O.time,eps=strainer.strain,sigma=strainer.stress)

def stopIfDamaged():
        currSig=plot.data['sigma'][-1] # last sigma value
        maxSig=max(plot.data['sigma']) # maximum sigma value
        # print something in any case, so that we know what is happening
        print plot.data['eps'][-1],currSig
        if currSig<.5*maxSig:
                print "Damaged, stopping"
                print 'gnuplot',plot.saveGnuplot(O.tags['id'])
                import sys
                sys.exit(0)

O.run(); O.wait()
# this place is never reached, since we call sys.exit(0) directly
```

Checkpoints

Occasionally, it is useful to revert to simulation at some past point and continue from it with different parameters. For instance, tension/compression test will use the same initial state but load it in 2 different directions. Two functions, Omega.saveTmp and Omega.loadTmp are provided for this purpose; *memory* is used as storage medium, which means that saving is faster, and also that the simulation will disappear when Yade finishes.

```
O.saveTmp()
# do something
O.saveTmp('foo')
O.loadTmp()      # loads the first state
O.loadTmp('foo') # loads the second state
```

Warning: `O.loadTmp` cannot be called from inside an engine, since *before* loading a simulation, the old one must finish the current iteration; it would lead to deadlock, since `O.loadTmp` would wait for the current iteration to finish, while the current iteration would be blocked on `O.loadTmp`.
A special trick must be used: a separate function to be run after the current iteration is defined and is invoked from an independent thread launched only for that purpose:

```
O.engines=[...,PyRunner('myFunc()',iterPeriod=345)]

def myFunc():
        if someCondition:
                import thread
                # the () are arguments passed to the function
                thread.start_new_thread(afterIterFunc,())
def afterIterFunc():
        O.pause(); O.wait() # wait till the iteration really finishes
        O.loadTmp()

O.saveTmp()
O.run()
```

4.2.3 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 using telnet:

```
$ telnet localhost 9000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Enter auth cookie: dcekyu
__   __    ____                  __  _____ ____ ____
\ \ / /_ _|  _ \  ___    ___   / / |_   _/ ___|  _ \
 \ V / _` | | | |/ _ \  / _ \ / /    | || |   | |_) |
  | | (_| | |_| |  __/ | (_) / /     | || |___|  __/
  |_|\__,_|____/ \___|  \___/_/      |_| \____|_|

(connected from 127.0.0.1:40372)
>>>
```

The python pseudo-prompt `>>>` lets you write commands to manipulate simulation in variety of ways as usual. Two things to notice:

1. The new python interpreter (`>>>`) lives in a namespace separate from `Yade [1]:` command-line. For your convenience, `from yade import *` is run in the new python instance first, but local and global variables are not accessible (only builtins are).
2. The (fake) `>>>` interpreter does not have rich interactive feature of IPython, which handles the usual command-line `Yade [1]:`; therefore, you will have no command history, `?` help and so on.

Note: By giving access to python interpreter, full control of the system (including reading user's files) is possible. For this reason, **connection are only allowed from localhost**, not over network remotely. Of course you can log into the system via SSH over network to get remote access.

Warning: Authentication cookie is trivial to crack via bruteforce attack. Although the listener stalls for 5 seconds after every failed login attempt (and disconnects), the cookie could be guessed by trial-and-error during very long simulations on a shared computer.

Info provider

`TCP Info provider` listens at port 21000 (or higher) and returns some basic information about current simulation upon connection; the connection terminates immediately afterwards. The information is python dictionary represented as string (serialized) using standard pickle module.

This functionality is used by the batch system (described below) to be informed about individual simulation progress and estimated times. If you want to access this information yourself, you can study core/main/yade-batch.in for details.

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

readParamsFromTable knows which parameter file and which line to read by inspecting the `PARAM_TABLE` environment variable, set by the batch system.

parameter table simple text file, each line representing one parameter set. This file is read by readParamsFromTable (using TableParamReader class), called from simulation script, as explained above. For better reading of the text file you can make use of tabulators, these will be ignored by readParamsFromTable. Parameters are not restricted to numerical values. You can also make use of strings by "quoting" them (' ' may also be used instead of " "). This can be useful for nominal parameters.

The batch can be run as

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

and it will intelligently run one simulation for each parameter table line. A minimal example is found in examples/test/batch/params.table and examples/test/batch/sim.py, another example follows.

Example

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

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

hi_rho_v       =    20
hi_rh0_lo_v    =     5
```

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

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

```
readParamsFromTable(
        gravity=-9.81,
        density=2400,
        initialVelocity=20,
        noTableOk=True     # use default values if not run in batch
)
from yade.params.table import *
print gravity, density, initialVelocity
```

after the call to readParamsFromTable, corresponding python variables are created in the `yade.params.table` module and can be readily used in the script, e.g.

```
GravityEngine(gravity=(0,0,gravity))
```

Let us see what happens when running the batch:

```
$ yade-batch batch.table batch.py
Will run `/usr/local/bin/yade-trunk' on `batch.py' with nice value 10, output redirected to `batch.@.log', 4 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. To add the used parameterset or if set the description of the job you could add O.tags['params'] to the filename.

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

```
saveGnuplot(O.tags['id'])
```

For larger simulations, it is advisable to create separate directory of that name first, putting all files inside afterwards:

```
os.mkdir(O.tags['id'])
O.engines=[
        # …
        VTKRecorder(fileName=O.tags['id']+'/'+'vtk'),
        # …
```

```
]
# …
O.saveGnuplot(O.tags['id']+'/'+'graph1')
```

Controlling parallel computation

Default total number of available cores is determined from `/proc/cpuinfo` (provided by Linux kernel); in addition, if `OMP_NUM_THREADS` environment variable is set, minimum of these two is taken. The `-j`/`--jobs` option can be used to override this number.

By default, each job uses all available cores for itself, which causes jobs to be effectively run in parallel. Number of cores per job can be globally changed via the `--job-threads` option.

Table column named `!OMP_NUM_THREADS` (`!` prepended to column generally means to assign *environment variable*, rather than python variable) controls number of threads for each job separately, if it exists.

If number of cores for a job exceeds total number of cores, warning is issued and only the total number of cores is used instead.

Merging gnuplot from individual jobs

Frequently, it is desirable to obtain single figure for all jobs in the batch, for comparison purposes. Somewhat 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 (even after waitIfBatch()) , which prints:

```
gnuplot 20100413T144723p7625.gnuplot
```

to the output (redirected to log file).

This file itself contains single graph:

At the end, the batch system knows about all gnuplot files and tries to merge them together, by assembling the `merged.gnuplot` file.

HTTP overview

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

4.3 Postprocessing

4.3.1 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.

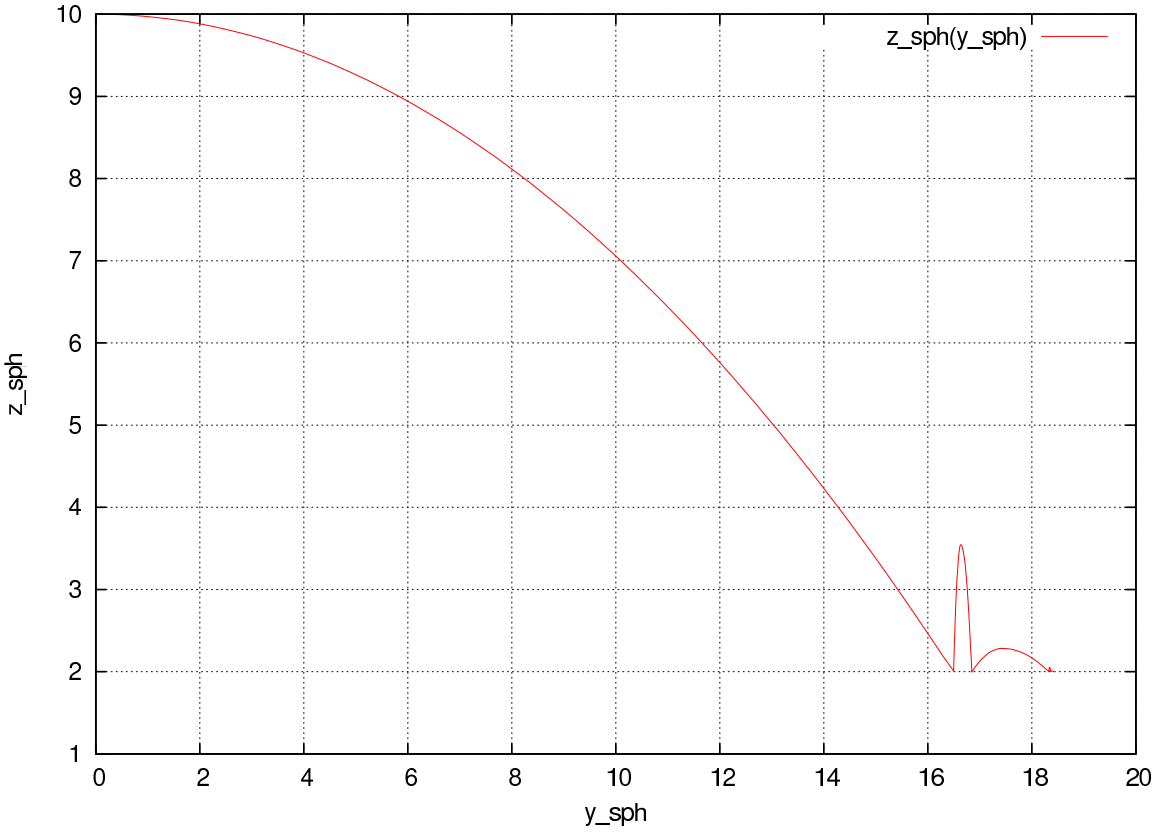


Figure 4.9: Figure from single job in the batch.

2. Periodic frame snapshot using SnapshotEngine (see examples/bulldozer/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 makeVideo:

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

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

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

Paraview

Saving data during the simulation

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

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

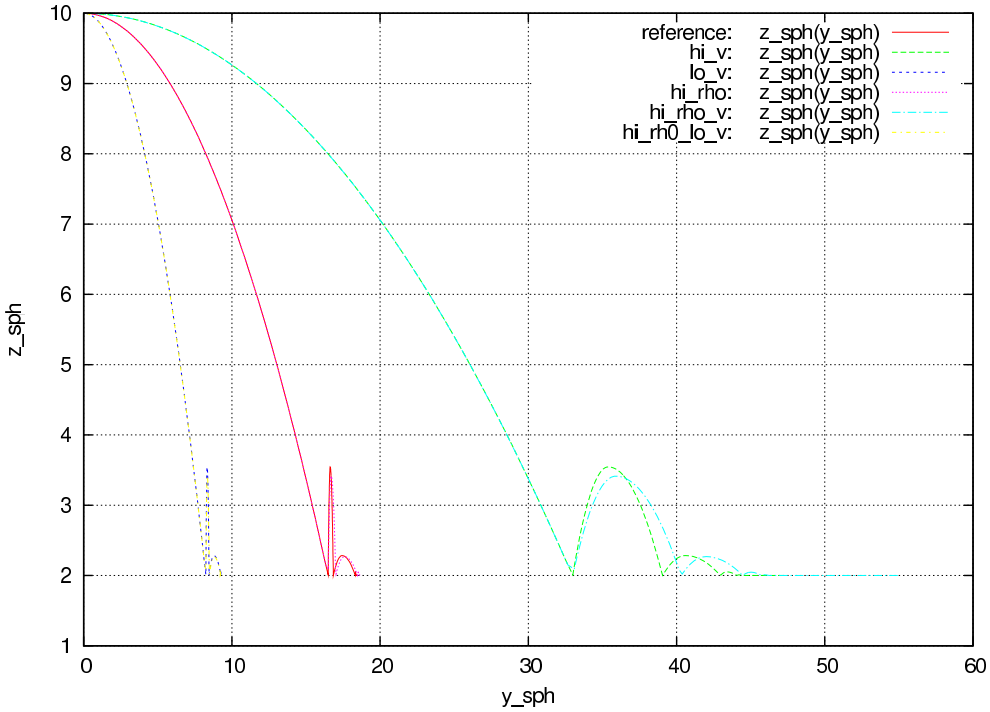


Figure 4.10: Merged figure from all jobs in the batch. Note that labels are prepended by job description to make lines distinguishable.

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

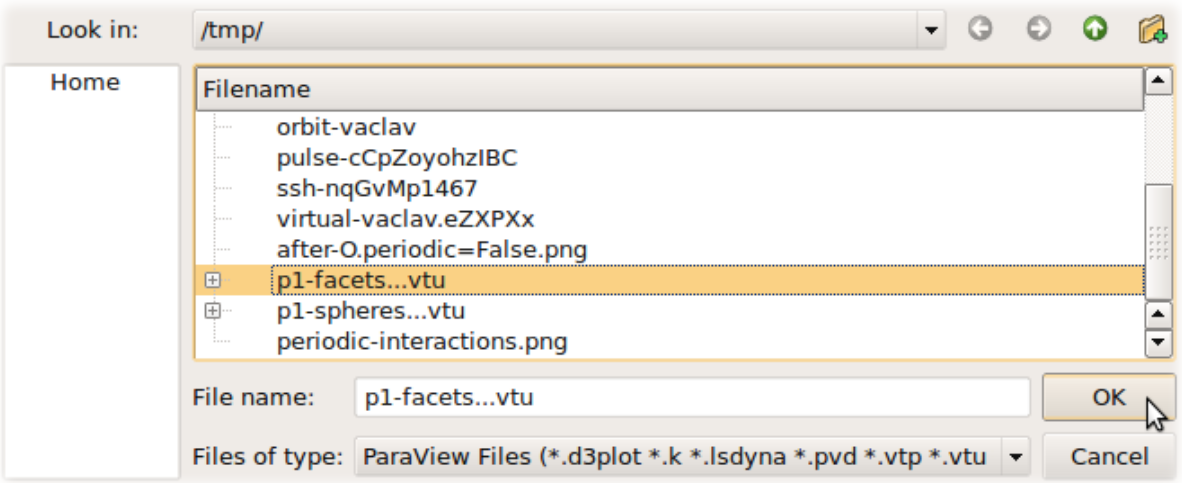
Figure 4.11: Summary page available at port 9080 as batch is processed (updates every 5 seconds automatically). Possible job statuses are pending, running, done, failed.

- 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

exporter.VTKExporter plays a similar role, with the difference that it is more flexible. It will save any user defined variable associated to the bodies.

Loading data into Paraview

All sets of files (`spheres`, `facets`, ...) must be opened one-by-one in Paraview. The open dialogue automatically collapses numbered files in one, making it easy to select all of them:

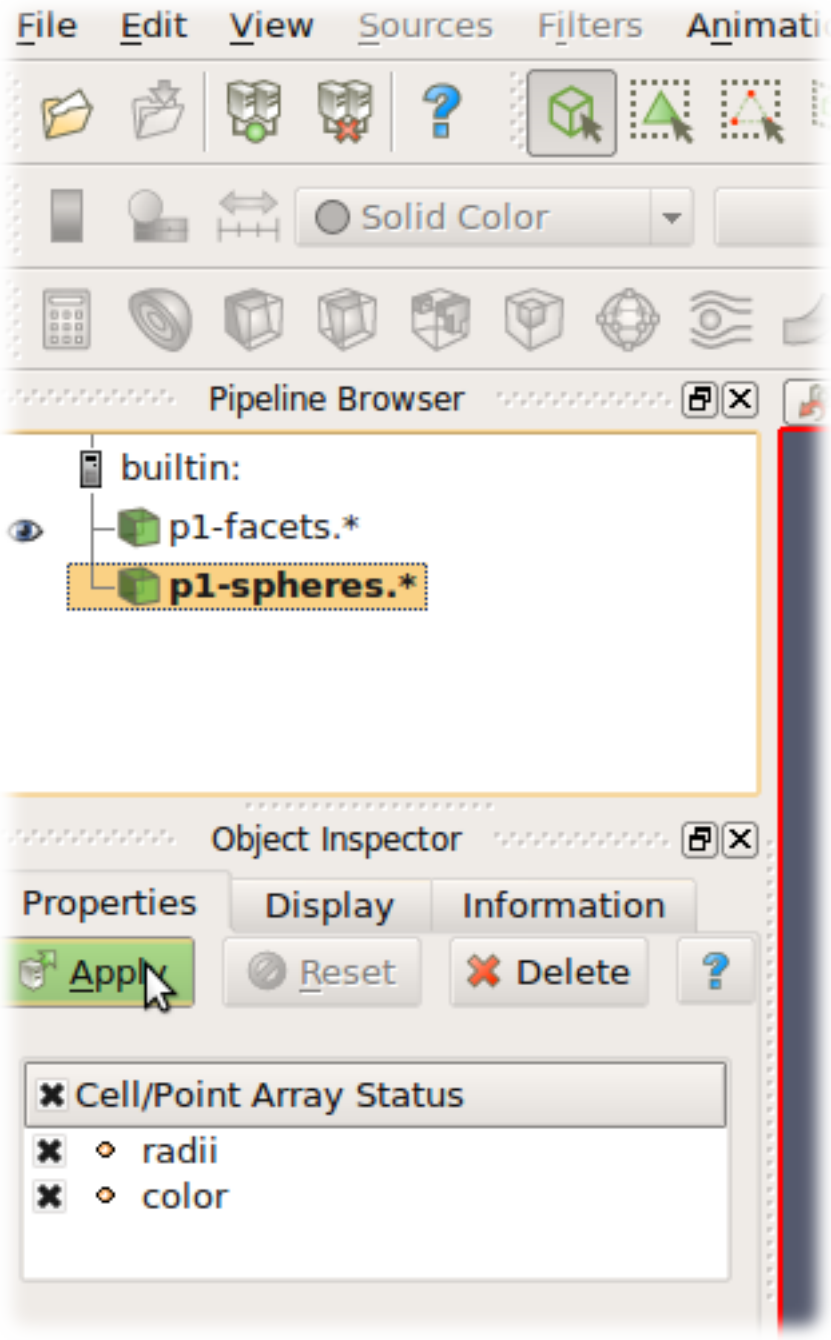


Click on the "Apply" button in the "Object inspector" sub-window to make loaded objects visible. You can see tree of displayed objects in the "Pipeline browser":

Rendering spherical particles. Glyphs Spheres will only appear as points. To make them look as spheres, you have to add "glyph" to the `p1-spheres.*` item in the pipeline using the  icon. Then set (in the Object inspector)

- "Glyph type" to *Sphere*
- "Radius" to *1*
- "Scale mode" to *Scalar* (*Scalar* is set above to be the *radii* value saved in the file, therefore spheres with radius *1* will be scaled by their true radius)
- "Set scale factor" to *1*
- optionally uncheck "Mask points" and "Random mode" (they make some particles not to be rendered for performance reasons, controlled by the "Maximum Number of Points")

After clicking "Apply", spheres will appear. They will be rendered over the original white points, which you can disable by clicking on the eye icon next to `p1-spheres.*` in the Pipeline browser.


File
Edit
View
Sources
Filters
Animati
Solid Color
Pipeline Browser
builtin:
p1-facets.*
p1-spheres.*
Object Inspector
Properties
Display
Information
Apply
Reset
Delete
Cell/Point Array Status
radii
color

Rendering spherical particles. PointSprite Another opportunity to display spheres is an using *PointSprite* plugin. This technique requires much less RAM in comparison to Glyphs.

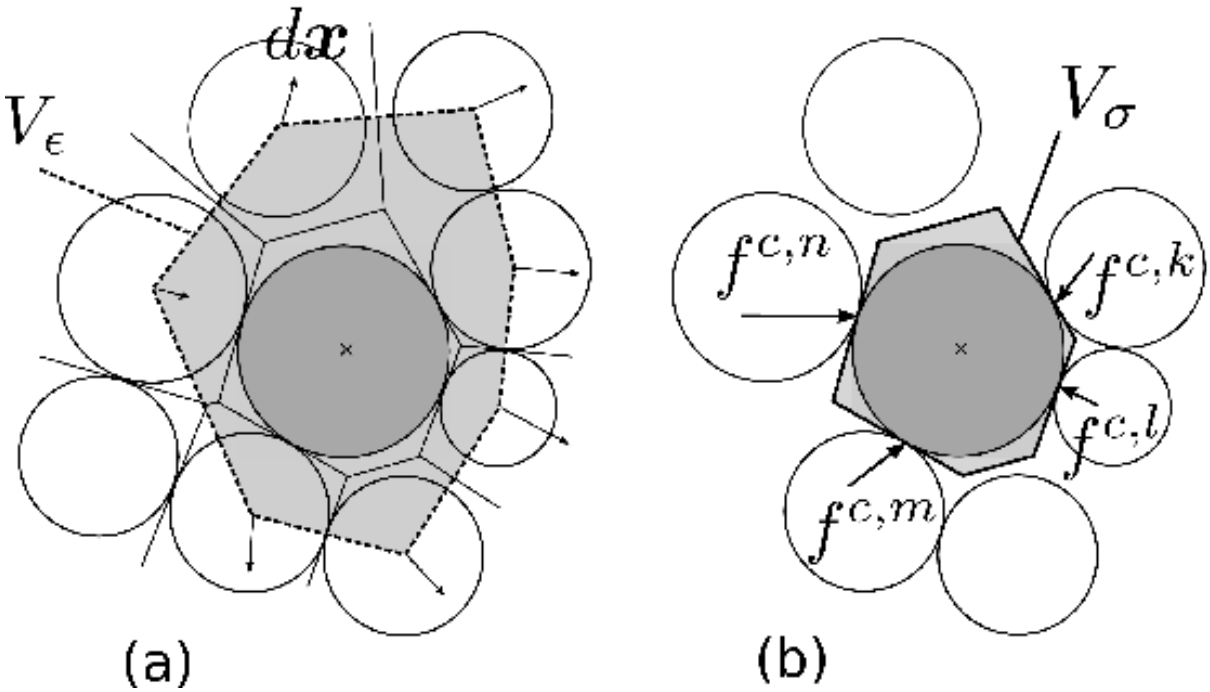
- "Tools -> Manage Plugins"
- "PointSprite_Plugin -> Load selected -> Close"
- Load VTU-files
- "Representation -> Point Sprite"
- "Point Sprite -> Scale By -> radii"
- "Edit Radius Transfer Function -> Proportional -> Multiplier = 1.0 -> Close"

Facet transparency If you want to make facet objects transparent, select `p1-facets.*` in the Pipeline browser, then go to the Object inspector on the Display tab. Under "Style", you can set the "Opacity" value to something smaller than 1.

Animation You can move between frames (snapshots that were saved) via the "Animation" menu. After setting the view angle, zoom etc to your satisfaction, the animation can be saved with *File/Save animation*.

4.3.2 Micro-stress and micro-strain

It is sometimes useful to visualize a DEM simulation through equivalent strain fields or stress fields. This is possible with TesselationWrapper. This class handles the triangulation of spheres in a scene, build tesselation on request, and give access to computed quantities: volume, porosity and local deformation for each sphere. The definition of microstrain and microstress is at the scale of particle-centered subdomains shown below, as explained in [Catalano2014a] .

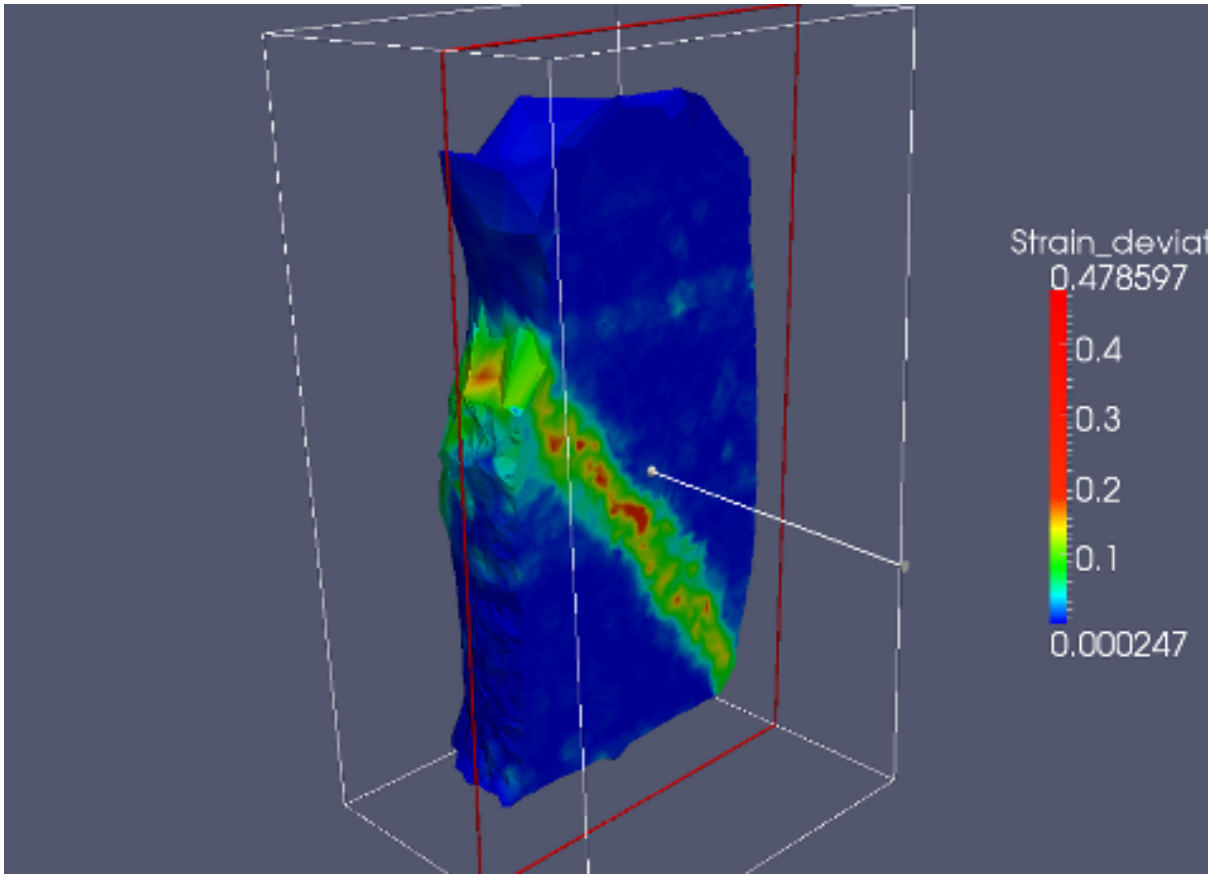


Micro-strain

Below is an output of the defToVtk function visualized with paraview (in this case Yade's TesselationWrapper was used to process experimental data obtained on sand by Edward Ando at Grenoble

University, 3SR lab.). The output is visualized with paraview, as explained in the previous section. Similar results can be generated from simulations:

```
tt=TriaxialTest()
tt.generate("test.yade")
O.load("test.yade")
O.run(100,True)
TW=TesselationWrapper()
TW.triangulate()        #compute regular Delaunay triangulation, don't construct tesselation
TW.computeVolumes()     #will silently tesselate the packing, then compute volume of each Voronoi cell
TW.volume(10)           #get volume associated to sphere of id 10
TW.setState(0)          #store current positions internaly for later use as the "0" state
O.run(100,True)         #make particles move a little (let's hope they will!)
TW.setState(1)          #store current positions internaly in the "1" (deformed) state
#Now we can define strain by comparing states 0 and 1, and average them at the particles scale
TW.defToVtk("strain.vtk")
```



Micro-stress

Stress fields can be generated by combining the volume returned by TesselationWrapper to per-particle stress given by bodyStressTensors. Since the stress σ from bodyStressTensor implies a division by the volume V_b of the solid particle, one has to re-normalize it in order to obtain the micro-stress as defined in [Catalano2014a] (equation 39 therein), i.e. $\overline{\sigma}^k = \sigma^k \times V_b^k / V_\sigma^k$ where V_σ^k is the volume assigned to particle k in the tesselation. For instance:

```
#"b" being a body
TW=TesselationWrapper()
TW.computeVolumes()
s=bodyStressTensors()
stress = s[b.id]**4.*pi/3.*b.shape.radius**3/TW.volume(b.id)
```

As any other value, the stress can be exported to a vtk file for display in Paraview using export.VTKExporter.

4.4 Python specialties and tricks

4.4.1 Importing Yade in other Python applications

Yade can be imported in other Python applications. To do so, you need somehow to make yade executable .py extended. The easiest way is to create a symbolic link, i.e. (suppose your Yade executable file is called "yade-trunk" and you want make it "yadeimport.py"):

```
$ cd /path/where/you/want/yadeimport
$ ln -s /path/to/yade/executable/yade-trunk yadeimport.py
```

Then you need to make your yadeimport.py findable by Python. You can export PYTHONPATH environment variable, or simply use sys.path directly in Python script:

```
import sys
sys.path.append('/path/where/you/want/yadeimport')
from yadeimport import *

print Matrix3(1,2,3, 4,5,6, 7,8,9)
print O.bodies
# any other Yade code
```

4.5 Extending Yade

- new particle shape
- new constitutive law

4.6 Troubleshooting

4.6.1 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.

4.6.2 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.

4.6.3 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 5

Parallel hierarchical multiscale modeling of granular media by coupling FEM and DEM with open-source codes Escript and YADE

Authors: Ning Guo and Jidong Zhao

Institution: Hong Kong University of Science and Technology

Escript download page: https://launchpad.net/escript-finley

mpi4py download page (optional, require MPI): https://bitbucket.org/mpi4py/mpi4py

Tested platforms: Desktop with Ubuntu 10.04, 32 bit; Server with Ubuntu 12.04, 14.04, 64 bit; Cluster with Centos 6.2, 6.5, 64 bit;

5.1 Introduction

The code is built upon two open source packages: Yade for DEM modules and Escript for FEM modules. It implements the hierarchical multiscale model (FEMxDEM) for simulating the boundary value problem (BVP) of granular media. FEM is used to discretize the problem domain. Each Gauss point of the FEM mesh is embedded a representative volume element (RVE) packing simulated by DEM which returns local material constitutive responses to FEM. Typically, hundreds to thousands of RVEs are involved in a medium-sized problem which is critically time consuming. Hence parallelization is achieved in the code through either multiprocessing on a supercomputer or mpi4py on a HPC cluster (require MPICH or Open MPI). The MPI implementation in the code is quite experimental. The "mpipool.py" is contributed by Lisandro Dalcin, the author of mpi4py package. Please refer to the examples for the usage of the code.

5.2 Work on the YADE side

The version of YADE should be at least rev3682 in which Bruno added the stringToScene function. Before installation, I added some functions to the source code (in "yade" subfolder). But only one function ("Shop::getStressAndTangent" in "./pkg/dem/Shop.cpp") is necessary for the FEMxDEM coupling, which returns the stress tensor and the tangent operator of a discrete packing. The former is homogenized using the Love's formula and the latter is homogenized as the elastic modulus. After installation and we get the executable file: yade-versionNo. We then generate a .py file linked to the executable

file by "ln yade-versionNo yadeimport.py". This .py file will serve as a wrapped library of YADE. Later on, we will import all YADE functions into the python script through "from yadeimport import *" (see simDEM.py file).

Open a python terminal. Make sure you can run

```
import sys
sys.path.append('where you put yadeimport.py')
from yadeimport import *
Omega().load('your initial RVE packing, e.g. 0.yade.gz')
```

If you are successful, you should also be able to run

```
from simDEM import *
```

5.3 Work on the Escript side

No particular requirement. But make sure the modules are callable in python, which means the main folder of Escript should be in your PYTHONPATH and LD_LIBRARY_PATH. The modules are wrapped as a class in msFEM*.py.

Open a python terminal. Make sure you can run:

```
from esys.escript import *
from esys.escript.linearPDEs import LinearPDE
from esys.finley import Rectangle
```

(Note: Escript is used for the current implementation. It can be replaced by any other FEM package provided with python bindings, e.g. FEniCS (http://fenicsproject.org). But the interface files "msFEM*.py" need to be modified.)

5.4 Example tests

After Steps 1 & 2, one should be able to run all the scripts for the multiscale analysis. The initial RVE packing (default name "0.yade.gz") should be provided by the user (e.g. using YADE to prepare a consolidated packing), which will be loaded by simDEM.py when the problem is initialized. The sample is initially uniform as long as the same RVE packing is assigned to all the Gauss points in the problem domain. It is also possible for the user to specify different RVEs at different Gauss points to generate an inherently inhomogeneous sample.

While simDEM.py is always required, only one msFEM*.py is needed for a single test. For example, in a 2D (3D) dry test, msFEM2D.py (msFEM3D.py) is needed; similarly for a coupled hydro-mechanical problem (2D only, saturated), msFEMup.py is used which incorporates the u-p formulation. Multiprocessing is used by default. To try MPI parallelization, please set useMPI=True when constructing the problem in the main script. Example tests given in the "example" subfolder are listed below. Note: The initial RVE packing (named 0.yade.gz by default) needs to be generated, e.g. using prepareRVE.py in "example" subfolder for a 2D packing (similarly for 3D).

1. **2D drained biaxial compression test on dry dense sand** (biaxialSmooth.py) *Note*: Test description and result were presented in [Guo2014] and [Guo2014c].
2. **2D passive failure under translational mode of dry sand retained by a rigid and frictionless wall** (retainingSmooth.py) *Note:* Rolling resistance model (CohFrictMat) is used in the RVE packing. Test description and result were presented in [Guo2015].
3. **2D half domain footing settlement problem with mesh generated by Gmsh** (footing.py, footing.msh) *Note:* Rolling resistance model (CohFrictMat) is used in the RVE packing. Six-node triangle element is generated by Gmsh with three Gauss points each. Test description and result were presented in [Guo2015].

4. **3D drained conventional triaxial compression test on dry dense sand using MPI parallelism** (triaxialRough.py) *Note 1:* The simulation is very time consuming. It costs ~4.5 days on one node using multiprocessing (16 processes, 2.0 GHz CPU). When useMPI is switched to True (as in the example script) and four nodes are used (80 processes, 2.2 GHz CPU), the simulation costs less than 24 hours. The speedup is about 4.4 in our test. *Note 2:* When MPI is used, mpi4py is required to be installed. The MPI implementation can be either MPICH or Open MPI. The file "mpipool.py" should also be placed in the main folder. Our test is based on openmpi-1.6.5. This is an on-going work. Test description and result will be presented later.
5. **2D globally undrained biaxial compression test on saturated dense sand with changing permeability using MPI parallelism** (undrained.py) *Note:* This is an on-going work. Test description and result will be presented later.

5.5 Disclaim

This work extensively utilizes and relies on some third-party packages as mentioned above. Their contributions are acknowledged. Feel free to use and redistribute the code. But there is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Chapter 6

Programmer's manual

6.1 Build system

Yade uses cmake the cross-platform, open-source build system for managing the build process. It takes care of configuration, compilation and installation. CMake is used to control the software compilation process using simple platform and compiler independent configuration files. CMake generates native makefiles and workspaces that can be used in the compiler environment of your choice.

6.1.1 Building

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

```
core/          ## core simulation building blocks
extra/         ## miscillanea
gui/           ## user interfaces
   qt4/           ## 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
```

Header installation

To allow flexibility in source layout, CMAKE 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:

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

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

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 CMAKE 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.

Linking

The order in which modules might depend on each other is given as follows:

module	resulting shared library	dependencies
lib	`libyade-support.so`	can depend on external libraries, may **not** depend on any other part of Yade.
core	`libcore.so`	`yade-support`; *may* depend on external libraries.
pkg	`libplugins.so`	`core`, `yade-support`
gui	`libQtGUI.so`, `libPythonUI.so`	`lib`, `core`, `pkg`
py	(many files)	`lib`, `core`, `pkg`, external

6.2 Development tools

6.2.1 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 of wiki pages is dedicated to these tools in the development section of the wiki.

6.2.2 Hosting and versioning

The Yade project is kindly hosted at launchpad, which is used for source code, bug tracking, planning, package downloads and more.

The versioning software used is GIT, for which a short tutorial can be found in *Yade on GitHub*. GIT is a distributed revision control system. It is available packaged for all major linux distributions.

The source code is hosted on GitHub , which is periodically imported to Launchpad for building PPA-packages. The repository can be http-browsed.

6.2.3 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 GIT, 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 formats. 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.

6.2.4 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 here. They differ with regression tests in the sense that they simulate more complex situations and combinations of different engines, and usually don't have a mathematical proof (though there is no restriction on the latest). They compare the values obtained in version N with values obtained in a previous version or any other "expected" results. The reference values must be hardcoded in the script itself or in data files provided with the script. Check tests are based on regular yade scripts, so that users can easily commit their own scripts to trunk in order to get some automatized testing after commits from other developers.

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.

6.3 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 `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)

6.3.1 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 [1] 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] Not considering OpenGL dispatchers, which might be replaced by regular virtual functions in the future.

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

To give a few examples:

- Bo1_Sphere_Aabb is a BoundFunctor which is called for Sphere, creating an instance of Aabb.
- Ig2_Facet_Sphere_ScGeom is binary functor called for Facet and Sphere, creating and instace of ScGeom.
- Law2_ScGeom_CpmPhys_Cpm is binary functor (LawFunctor) called for types ScGeom (Geom) and CpmPhys.

6.3.2 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`.

Sphinx documentation

Most c++ classes are wrapped in Python, which provides good introspection and interactive documentation (try writing `Material?` in the ipython prompt; or `help(CpmState)`).

Syntax of documentation is ReST (reStructuredText, see reStructuredText Primer). It is the same for c++ and python code.

- Documentation of c++ classes exposed to python is given as 3rd argument to *YADE_CLASS_BASE_DOC_* macro family* introduced below.
- Python classes/functions are documented using regular python docstrings. Besides explaining functionality, meaning and types of all arguments should also be documented. Short pieces of code might be very helpful. See the utils module for an example.

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).

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.

6.4 Support framework

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

6.4.1 Pointers

Shared pointers

Yade makes extensive use of shared pointers `shared_ptr`. [2] Although it probably has some performance impacts, it greatly simplifies memory management, ownership management of c++ objects in python and so forth. To obtain raw pointer from a `shared_ptr`, use its `get()` method; raw pointers should be used in case the object will be used only for short time (during a function call, for instance) and not stored anywhere.

Python defines thin wrappers for most c++ Yade classes (for all those registered with *YADE_CLASS_BASE_DOC_* macro family* and several others), which can be constructed from `shared_ptr`; in this way, Python reference counting blends with the `shared_ptr` reference counting model, preventing crashes due to python objects pointing to c++ objects that were destructed in the meantime.

Typecasting

Frequently, pointers have to be typecast; there is choice between static and dynamic casting.

- `dynamic_cast` (`dynamic_pointer_cast` for a `shared_ptr`) assures cast admissibility by checking runtime type of its argument and returns NULL if the cast is invalid; such check obviously costs time. Invalid cast is easily caught by checking whether the pointer is NULL or not; even if such check (e.g. `assert`) is absent, dereferencing NULL pointer is easily spotted from the stacktrace (debugger output) after crash. Moreover, `shared_ptr` checks that the pointer is non-NULL before dereferencing in debug build and aborts with "Assertion 'px!=0' failed." if the check fails.
- `static_cast` is fast but potentially dangerous (`static_pointer_cast` for `shared_ptr`). Static cast will return non-NULL pointer even if types don't allow the cast (such as casting from `State*` to `Material*`); the consequence of such cast is interpreting garbage data as instance of the class cast to, leading very likely to invalid memory access (segmentation fault, "crash" for short).

To have both speed and safety, Yade provides 2 macros:

`YADE_CAST` expands to `static_cast` in optimized builds and to `dynamic_cast` in debug builds.

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

[2] 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.

6.4.2 Basic numerics

The floating point type to use in Yade `Real`, which is by default typedef for `double`. [3]

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.

```
# cross product
Yade [2]: Vector3(1,2,3).cross(Vector3(0,0,1))
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/usr/lib/x86_64-linux-gnu/yadedaily/py/yade/__init__.pyc in <module>()
----> 1 Vector3(1,2,3).cross(Vector3(0,0,1))

NameError: name 'Vector3' is not defined

# construct quaternion from axis and angle
Yade [3]: Quaternion(Vector3(0,0,1),pi/2)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/usr/lib/x86_64-linux-gnu/yadedaily/py/yade/__init__.pyc in <module>()
----> 1 Quaternion(Vector3(0,0,1),pi/2)

NameError: name 'Quaternion' is not defined
```

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.

6.4.3 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):

[3] 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.

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

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

Some RTTI information can be accessed from python:

```
Yade [2]: yade.system.childClasses('Shape')
Out[2]:
{'Box',
 'ChainedCylinder',
 'Clump',
 'Cylinder',
 'Facet',
 'GridConnection',
 'GridNode',
 'Polyhedra',
 'Sphere',
 'Tetra',
 'Wall'}

Yade [3]: Sphere().name            ## getClassName()
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/usr/lib/x86_64-linux-gnu/yadedaily/py/yade/__init__.pyc in <module>()
----> 1 Sphere().name            ## getClassName()

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

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

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

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

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

```
Yade [1]: O.engines=[GravityEngine(gravity=Vector3(0,0,-9.81))]
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/usr/lib/x86_64-linux-gnu/yadedaily/py/yade/__init__.pyc in <module>()
----> 1 O.engines=[GravityEngine(gravity=Vector3(0,0,-9.81))]
```

```
NameError: name 'Vector3' is not defined

Yade [2]: O.save('abc.xml')
```

and the XML looks like this:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<!DOCTYPE boost_serialization>
<boost_serialization signature="serialization::archive" version="10">
<scene class_id="0" tracking_level="0" version="1">
        <px class_id="1" tracking_level="1" version="0" object_id="_0">
                <Serializable class_id="2" tracking_level="1" version="0" object_id="_1"></Serializable>
                <dt>1e-08</dt>
                <iter>0</iter>
                <subStepping>0</subStepping>
                <subStep>-1</subStep>
                <time>0</time>
                <speed>0</speed>
                <stopAtIter>0</stopAtIter>
                <stopAtTime>0</stopAtTime>
                <isPeriodic>0</isPeriodic>
                <trackEnergy>0</trackEnergy>
                <doSort>0</doSort>
                <runInternalConsistencyChecks>1</runInternalConsistencyChecks>
                <selectedBody>-1</selectedBody>
                <flags>0</flags>
                <tags class_id="3" tracking_level="0" version="0">
                        <count>5</count>
                        <item_version>0</item_version>
                        <item>author=~(bchareyre@dt-rv020)</item>
                        <item>isoTime=20151119T215459</item>
                        <item>id=20151119T215459p31818</item>
                        <item>d.id=20151119T215459p31818</item>
                        <item>id.d=20151119T215459p31818</item>
                </tags>
                <engines class_id="4" tracking_level="0" version="0">
                        <count>0</count>
                        <item_version>1</item_version>
                </engines>
                <_nextEngines>
                        <count>0</count>
                        <item_version>1</item_version>
                </_nextEngines>
                <bodies class_id="5" tracking_level="0" version="1">
                        <px class_id="6" tracking_level="1" version="0" object_id="_2">
                                <Serializable object_id="_3"></Serializable>
                                <body class_id="7" tracking_level="0" version="0">
                                        <count>0</count>
                                        <item_version>1</item_version>
                                </body>
                        </px>
                </bodies>
                <interactions class_id="8" tracking_level="0" version="1">
                        <px class_id="9" tracking_level="1" version="0" object_id="_4">
                                <Serializable object_id="_5"></Serializable>
                                <interaction class_id="10" tracking_level="0" version="0">
                                        <count>0</count>
                                        <item_version>1</item_version>
                                </interaction>
                                <serializeSorted>0</serializeSorted>
                                <dirty>1</dirty>
                        </px>
                </interactions>
```

```
<energy class_id="11" tracking_level="0" version="1">
        <px class_id="12" tracking_level="1" version="0" object_id="_6">
                <Serializable object_id="_7"></Serializable>
                <energies class_id="13" tracking_level="0" version="0">
                        <size>0</size>
                </energies>
                <names class_id="14" tracking_level="0" version="0">
                        <count>0</count>
                        <item_version>0</item_version>
                </names>
                <resetStep>
                        <count>0</count>
                </resetStep>
        </px>
</energy>
<materials class_id="16" tracking_level="0" version="0">
        <count>0</count>
        <item_version>1</item_version>
</materials>
<bound class_id="17" tracking_level="0" version="1">
        <px class_id="-1"></px>
</bound>
<cell class_id="19" tracking_level="0" version="1">
        <px class_id="20" tracking_level="1" version="0" object_id="_8">
                <Serializable object_id="_9"></Serializable>
                <trsf class_id="21" tracking_level="0" version="0">
                        <m00>1</m00>
                        <m01>0</m01>
                        <m02>0</m02>
                        <m10>0</m10>
                        <m11>1</m11>
                        <m12>0</m12>
                        <m20>0</m20>
                        <m21>0</m21>
                        <m22>1</m22>
                </trsf>
                <refHSize>
                        <m00>1</m00>
                        <m01>0</m01>
                        <m02>0</m02>
                        <m10>0</m10>
                        <m11>1</m11>
                        <m12>0</m12>
                        <m20>0</m20>
                        <m21>0</m21>
                        <m22>1</m22>
                </refHSize>
                <hSize>
                        <m00>1</m00>
                        <m01>0</m01>
                        <m02>0</m02>
                        <m10>0</m10>
                        <m11>1</m11>
                        <m12>0</m12>
                        <m20>0</m20>
                        <m21>0</m21>
                        <m22>1</m22>
                </hSize>
                <prevHSize>
                        <m00>1</m00>
                        <m01>0</m01>
                        <m02>0</m02>
                        <m10>0</m10>
```

```
                                        <m11>1</m11>
                                        <m12>0</m12>
                                        <m20>0</m20>
                                        <m21>0</m21>
                                        <m22>1</m22>
                                </prevHSize>
                                <velGrad>
                                        <m00>0</m00>
                                        <m01>0</m01>
                                        <m02>0</m02>
                                        <m10>0</m10>
                                        <m11>0</m11>
                                        <m12>0</m12>
                                        <m20>0</m20>
                                        <m21>0</m21>
                                        <m22>0</m22>
                                </velGrad>
                                <nextVelGrad>
                                        <m00>0</m00>
                                        <m01>0</m01>
                                        <m02>0</m02>
                                        <m10>0</m10>
                                        <m11>0</m11>
                                        <m12>0</m12>
                                        <m20>0</m20>
                                        <m21>0</m21>
                                        <m22>0</m22>
                                </nextVelGrad>
                                <prevVelGrad>
                                        <m00>0</m00>
                                        <m01>0</m01>
                                        <m02>0</m02>
                                        <m10>0</m10>
                                        <m11>0</m11>
                                        <m12>0</m12>
                                        <m20>0</m20>
                                        <m21>0</m21>
                                        <m22>0</m22>
                                </prevVelGrad>
                                <homoDeform>1</homoDeform>
                                <velGradChanged>0</velGradChanged>
                        </px>
                </cell>
                <miscParams class_id="22" tracking_level="0" version="0">
                        <count>0</count>
                        <item_version>1</item_version>
                </miscParams>
                <dispParams class_id="23" tracking_level="0" version="0">
                        <count>0</count>
                        <item_version>1</item_version>
                </dispParams>
        </px>
</scene>
</boost_serialization>
```

Warning: Since XML files closely reflect implementation details of Yade, they will not be compatible between different versions. Use them only for short-term saving of scenes. Python is *the* high-level description Yade uses.

Python attribute access

The macro *YADE_CLASS_BASE_DOC_* macro family* introduced above is (behind the scenes) also used to create functions for accessing attributes from Python. As already noted, set of serialized attributes and set of attributes accessible from Python are identical. Besides attribute access, these wrapper classes imitate also some functionality of regular python dictionaries:

```
Yade [1]: s=Sphere()

Yade [2]: s.radius              ## read-access
Out[2]: nan

Yade [3]: s.radius=4.           ## write access

Yade [4]: s.dict().keys()               ## show all available keys
Out[4]: ['color', 'highlight', 'wire', 'radius']

Yade [5]: for k in s.dict().keys(): print s.dict()[k]  ## iterate over keys, print their values
   ...:
Vector3(1,1,1)
False
False
4.0

Yade [6]: s.dict()['radius']             ## same as: 'radius' in s.keys()
Out[6]: 4.0

Yade [7]: s.dict()                       ## show dictionary of both attributes and values
Out[7]: {'color': Vector3(1,1,1), 'highlight': False, 'radius': 4.0, 'wire': False}
```

6.4.5 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++.

6.4.6 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 currenly 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_ScGeom: public IGeomFunctor{
   public:

   // override the IGeomFunctor::go
   //   (it is really inherited from FunctorWrapper template,
   //    therefore not declare explicitly in the
   //    IGeomFunctor declaration as such)
   // since dispatcher dispatches only for declared types
   //   (or types derived from them), we can do
   //   static_cast<Facet>(shape1) and static_cast<Sphere>(shape2)
   //   in the ::go body, without worrying about types being wrong.
   virtual bool go(
      // objects for dispatch
      const shared_ptr<Shape>& shape1, const shared_ptr<Shape>& shape2,
      // other arguments
      const State& state1, const State& state2, const Vector3r& shift2,
      const bool& force, const shared_ptr<Interaction>& c
   );
   /* … */

   // this declares the type we want to be dispatched for, matching
   //   first 2 arguments to ::go and first 2 classes in TYPELIST_7 above
   //   shape1 is a Facet and shape2 is a Sphere
   //   (or vice versa, see lines below)
   FUNCTOR2D(Facet,Sphere);

   // declare how to swap the arguments
   //   so that we can receive those as well
   DEFINE_FUNCTOR_ORDER_2D(Facet,Sphere);
   /* … */
};
```

Dispatch resolution

The dispatcher doesn't always have functors that exactly match the actual types it receives. In the same way as virtual methods, it tries to find the closest match in such way that:

1. the actual instances are derived types of those the functor accepts, or exactly the accepted types;
2. sum of distances from actual to accepted types is sharp-minimized (each step up in the class hierarchy counts as 1)

If no functor is able to accept given types (first condition violated) or multiple functors have the same distance (in condition 2), an exception is thrown.

This resolution mechanism makes it possible, for instance, to have a hierarchy of ScGeom classes (for different combination of shapes), but only provide a LawFunctor accepting `ScGeom`, rather than having different laws for each shape combination.

Note: Performance implications of dispatch resolution are relatively low. The dispatcher lookup is only done once, and uses fast lookup matrix (1D or 2D); then, the functor found for this type(s) is cached within the `Interaction` (or `Body`) instance. Thus, regular functor call costs the same as dereferencing pointer and calling virtual method. There is blueprint to avoid virtual function call as well.

Note: At the beginning, the dispatch matrix contains just entries exactly matching given functors. Only when necessary (by passing other types), appropriate entries are filled in as well.

Indexing dispatch types

Classes entering the dispatch mechanism must provide for fast identification of themselves and of their parent class. [4] 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);
};
```

At runtime, each class within the top-level Indexable hierarchy has its own unique numerical index. These indices serve to build the dispatch matrix for each dispatcher.

Inspecting dispatch in python

If there is a need to debug/study multiple dispatch, python provides convenient interface for this low-level functionality.

We can inspect indices with the `dispIndex` property (note that the top-level indexable `Shape` has negative (invalid) class index; we purposively didn't call `createIndex` in its constructor):

```
Yade [1]: Sphere().dispIndex, Facet().dispIndex, Wall().dispIndex
Out[1]: (1, 5, 10)
```

[4] 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.

```
Yade [2]: Shape().dispIndex                    # top-level indexable
Out[2]: -1
```

Dispatch hierarchy for a particular class can be shown with the `dispHierarchy()` function, returning list of class names: 0th element is the instance itself, last element is the top-level indexable (again, with invalid index); for instance:

```
Yade [1]: ScGeom().dispHierarchy()       # parent class of all other ScGeom_ classes
Out[1]: ['ScGeom', 'GenericSpheresContact', 'IGeom']

Yade [2]: ScGridCoGeom().dispHierarchy(), ScGeom6D().dispHierarchy(), CylScGeom().dispHierarchy()
Out[2]:
(['ScGridCoGeom', 'ScGeom6D', 'ScGeom', 'GenericSpheresContact', 'IGeom'],
 ['ScGeom6D', 'ScGeom', 'GenericSpheresContact', 'IGeom'],
 ['CylScGeom', 'ScGeom', 'GenericSpheresContact', 'IGeom'])

Yade [3]: CylScGeom().dispHierarchy(names=False)   # show numeric indices instead
Out[3]: [4, 1, 0, -1]
```

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

```
Yade [1]: ig=IGeomDispatcher([
   ...:     Ig2_Sphere_Sphere_ScGeom(),
   ...:     Ig2_Facet_Sphere_ScGeom(),
   ...:     Ig2_Wall_Sphere_ScGeom()
   ...:  ])
   ...:

Yade [2]: ig.dispMatrix()
Out[2]:
{('Facet', 'Sphere'): 'Ig2_Facet_Sphere_ScGeom',
 ('Sphere', 'Facet'): 'Ig2_Facet_Sphere_ScGeom',
 ('Sphere', 'Sphere'): 'Ig2_Sphere_Sphere_ScGeom',
 ('Sphere', 'Wall'): 'Ig2_Wall_Sphere_ScGeom',
 ('Wall', 'Sphere'): 'Ig2_Wall_Sphere_ScGeom'}

Yade [3]: ig.dispMatrix(False)          # don't convert to class names
Out[3]:
{(1, 1): 'Ig2_Sphere_Sphere_ScGeom',
 (1, 5): 'Ig2_Facet_Sphere_ScGeom',
 (1, 10): 'Ig2_Wall_Sphere_ScGeom',
 (5, 1): 'Ig2_Facet_Sphere_ScGeom',
 (10, 1): 'Ig2_Wall_Sphere_ScGeom'}
```

We can see that functors make use of symmetry (i.e. that Sphere+Wall are dispatched to the same functor as Wall+Sphere).

Finally, dispatcher can be asked to return functor suitable for given argument(s):

```
Yade [1]: ld=LawDispatcher([Law2_ScGeom_CpmPhys_Cpm()])

Yade [2]: ld.dispMatrix()
Out[2]: {('GenericSpheresContact', 'CpmPhys'): 'Law2_ScGeom_CpmPhys_Cpm'}

# see how the entry for ScGridCoGeom will be filled after this request
Yade [3]: ld.dispFunctor(ScGridCoGeom(),CpmPhys())
Out[3]: <Law2_ScGeom_CpmPhys_Cpm instance at 0x3491720>

Yade [4]: ld.dispMatrix()
Out[4]:
{('GenericSpheresContact', 'CpmPhys'): 'Law2_ScGeom_CpmPhys_Cpm',
 ('ScGridCoGeom', 'CpmPhys'): 'Law2_ScGeom_CpmPhys_Cpm'}
```

OpenGL functors

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

```
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:

```
Yade [2]: yade.system.childClasses('GlShapeFunctor')
Out[2]:
{'Gl1_Box',
 'Gl1_ChainedCylinder',
 'Gl1_Cylinder',
 'Gl1_Facet',
 'Gl1_GridConnection',
 'Gl1_Polyhedra',
 'Gl1_Sphere',
 'Gl1_Tetra',
 'Gl1_Wall'}
```

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

6.4.7 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.

6.4.8 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:

```
Yade [1]: TriaxialTest(numberOfGrains=100).load()

Yade [2]: O.engines[0].label='firstEngine'   ## labeled engines will show by labels in the stats table

Yade [3]: import yade.timing;
```

```
Yade [4]: O.timingEnabled=True

Yade [5]: yade.timing.reset()                    ## not necessary if used for the first time

Yade [6]: O.run(50); O.wait()

Yade [7]: yade.timing.stats()
Name                                                Count                Time            Rel. time
-------------------------------------------------------------------------------------------------------
"firstEngine"                                          50                46us                0.62%
InsertionSortCollider                                  26              2027us               26.76%
InteractionLoop                                        50              3560us               46.99%
GlobalStiffnessTimeStepper                              2                15us                0.21%
TriaxialCompressionEngine                              50               550us                7.27%
TriaxialStateRecorder                                   3               170us                2.25%
NewtonIntegrator                                       50              1204us               15.90%
TOTAL                                                                  7577us              100.00%
```

Exec count and time can be accessed and manipulated through `Engine::timingInfo` from c++ or `Engine().execCount` and `Engine().execTime` properties in Python.

In-engine and in-functor timing

Timing within engines (and functors) is based on TimingDeltas class. It is made for timing loops (functors' loop is in their respective dispatcher) and stores 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_ScGeom_CpmPhys_Cpm: public LawFunctor {
   /* … */
   YADE_CLASS_BASE_DOC_ATTRS_CTOR(Law2_ScGeom_CpmPhys_Cpm,LawFunctor,"docstring",
      /* attrs */,
      /* constructor */
      timingDeltas=shared_ptr<TimingDeltas>(new TimingDeltas); // timingDeltas object is automaticall
   );
   // ...
};
```

2. Inside the loop, start the timing by calling `timingDeltas->start();`
3. At places of interest, call `timingDeltas->checkpoint("label")`. The label is used only for post-processing, data are stored based on the checkpoint position, not the label.

> **Warning:** Checkpoints must be always reached in the same order, otherwise the timing data will be garbage. Your code can still branch, but you have to put checkpoints to places which are in common.

```
void Law2_ScGeom_CpmPhys_Cpm::go(shared_ptr<IGeom>& _geom,
                                 shared_ptr<IPhys>& _phys,
                                 Interaction* I,
                                 Scene* scene)
{
   timingDeltas->start();                        // the point at which the first timing starts
   // prepare some variables etc here
   timingDeltas->checkpoint("setup");
```

```
    // find geometrical data (deformations) here
    timingDeltas->checkpoint("geom");
    // compute forces here
    timingDeltas->checkpoint("material");
    // apply forces, cleanup here
    timingDeltas->checkpoint("rest");
}
```

4. **Alternatively, you can compile Yade using -DCMAKE_CXX_FLAGS="-DUSE_TIMING_DELTA**

```
void Law2_ScGeom_CpmPhys_Cpm::go(shared_ptr<IGeom>& _geom,
                                 shared_ptr<IPhys>& _phys,
                                 Interaction* I,
                                 Scene* scene)
{
    TIMING_DELTAS_START();
    // prepare some variables etc here
    TIMING_DELTAS_CHECKPOINT("setup")
    // find geometrical data (deformations) here
    TIMING_DELTAS_CHECKPOINT("geom")
    // compute forces here
    TIMING_DELTAS_CHECKPOINT("material")
    // apply forces, cleanup here
    TIMING_DELTAS_CHECKPOINT("rest")
}
```

The output might look like this (note that functors are nested inside dispatchers and `TimingDeltas` inside their engine/functor):

```
Name                                 Count                 Time          Rel. time
-------------------------------------------------------------------------------
ForceReseter                           400               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_ScGeom_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 2/3 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).

6.4.9 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.

6.5 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.

6.5.1 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.

6.5.2 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:

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

Yade [3]: O.bodies.erase(1)
Out[3]: True

Yade [4]: [b.id for b in O.bodies]
Out[4]: [0, 2]
```

In loops parallelized using OpenMP, the loop must traverse integer interval (rather than using iterators):

```
const long size=(long)bodies.size();       // store this value, since it doesn't change during the loop
#pragma omp parallel for
for(long _id=0; _id<size; _id++){
   const shared_ptr<Body>& b(bodies[_id]);
   if(!b) continue;
   /* … */
}
```

6.5.3 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;
   /* … */
}
```

6.5.4 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:

```
Yade [1]: O.bodies.append(Body())
Out[1]: 3

Yade [2]: O.forces.addF(0,Vector3(1,2,3))
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/usr/lib/x86_64-linux-gnu/yadedaily/py/yade/__init__.pyc in <module>()
----> 1 O.forces.addF(0,Vector3(1,2,3))

NameError: name 'Vector3' is not defined

Yade [3]: O.forces.f(0)
Out[3]: Vector3(0,0,0)

Yade [4]: O.forces.f(100)
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/usr/lib/x86_64-linux-gnu/yadedaily/py/yade/__init__.pyc in <module>()
----> 1 O.forces.f(100)

IndexError: Body id out of range.
```

6.5.5 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 takes care of erasing those interactions that are no longer active (such as if bodies get too far apart for non-cohesive laws; or in case of complete damage for damage models). This is triggered by the LawFunctor returning false. For this reason it is of upmost importance for the LawFunctor to return consistently.

Such interaction will not be deleted immediately, but will be reset to potential state. At the next execution of the collider `InteractionContainer::conditionalyEraseNonReal` will be called, which will completely erase interactions only if the bounding boxes ceased to overlap; the rest will be kept in potential state.

Creating interactions explicitly

Interactions may still be created explicitly with utils.createInteraction, without any spatial requirements. This function searches current engines for dispatchers and uses them. IGeomFunctor is called with the `force` parameter, obliging it to return `true` even if there is no spatial overlap.

6.5.6 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.

6.6 Runtime structure

6.6.1 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 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.

6.6.2 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 [5] 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.

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

6.6.3 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().

6.7 Python framework

6.7.1 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:

```
Yade [1]: f1=ForceEngine()

Yade [2]: f1.ids=[0,4,5]

Yade [3]: f1.force=Vector3(0,-1,-2)
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/usr/lib/x86_64-linux-gnu/yadedaily/py/yade/__init__.pyc in <module>()
----> 1 f1.force=Vector3(0,-1,-2)

NameError: name 'Vector3' is not defined
```

and

```
Yade [1]: f2=ForceEngine(ids=[0,4,5],force=Vector3(0,-1,-2))
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/usr/lib/x86_64-linux-gnu/yadedaily/py/yade/__init__.pyc in <module>()
----> 1 f2=ForceEngine(ids=[0,4,5],force=Vector3(0,-1,-2))
```

```
NameError: name 'Vector3' is not defined

Yade [2]: print f1.dict()
{'ompThreads': -1, 'force': Vector3(0,0,0), 'ids': [0, 4, 5], 'dead': False, 'label': ''}

Yade [3]: print f2.dict()
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/usr/lib/x86_64-linux-gnu/yadedaily/py/yade/__init__.pyc in <module>()
----> 1 print f2.dict()

NameError: name 'f2' is not defined
```

Wrapped classes also inherit from Serializable several special virtual methods: dict() returning all registered class attributes as dictionary (shown above), clone() returning copy of instance (by copying attribute values), updateAttrs() and updateExistingAttrs() assigning attributes from given dictionary (the former thrown for unknown attribute, the latter doesn't).

Read-only property `name` wraps c++ method `getClassName()` returning class name as string. (Since c++ class and the wrapper class always have the same name, getting python type using `__class__` and its property `__name__` will give the same value).

```
Yade [1]: s=Sphere()

Yade [2]: s.__class__.__name__
Out[2]: 'Sphere'
```

6.7.2 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: 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.

6.7.3 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.

6.7.4 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; [6]

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.

6.8 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.

6.8.1 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>
   ```

[6] 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. On systems, where minieigen is available as a separate package, the Yade's miniEigen is skipped.

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.

6.8.2 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.

6.9 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 pripority than lower versions/revisions.

6.9.1 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).

6.9.2 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 7

Yade on GitHub

7.1 Fast checkout without GitHub account (read-only)

Getting the source code without registering on GitHub can be done via a single command. It will not allow interactions with the remote repository, which you access the read-only way:

```
git clone https://github.com/yade/trunk.git
```

7.2 Using branches on GitHub (for frequent commits see git/trunk section below)

Most usefull commands are below. For more details, see for instance http://gitref.org/index.html and https://help.github.com/articles/set-up-git

7.2.1 Setup

1. Register on github.com
2. Add your SSH key to GitHub:

   On the GitHub site Click "Account Settings" (top right) > Click "SSH keys" > Click "Add SSH key"
3. Set your username and email through terminal:

   ```
   git config --global user.name "Firstname Lastname"
   git config --global user.email "your_email@youremail.com"
   ```
4. Fork a repo:

   Click the "Fork" button on the https://github.com/yade/trunk
5. Set Up Your Local Repo through terminal:

   ```
   git clone git@github.com:username/trunk.git
   ```

   This creates a new folder, named trunk, that contains the whole code.
6. Configure remotes

   ```
   cd to/newly/created/folder
   git remote add upstream git@github.com:yade/trunk.git
   git fetch upstream
   ```

Now, your "trunk" folder is linked with the code hosted on github.com. Through appropriate commands explained below, you will be able to update your code to include changes commited by others, or to commit yourself changes that others can get.

7.2.2 Retrieving older Commits

In case you want to work with, or compile, an older version of Yade which is not tagged, you can create your own (local) branch of the corresponding daily build. Look here for details.

7.2.3 Committing and updating

For those used to other version control systems, note that the commit mechanisms in Git significantly differs from that of Bazaar or SVN. Therefore, don't expect to find a one-to-one command replacement. In some cases, however, the equivalent bazaar command is indicated below to ease the transition.

Inspecting changes

You may start by inspecting your changes with a few commands. For the "diff" command, it is convenient to copy from the output of "status" instead of typing the path to modified files.

```
git status
git diff path/to/modified/file.cpp
```

Committing changes

Then you proceed to commit through terminal:

```
git add path/to/new/file.cpp  #Version a newly created file: equivalent of "bzr add"
git commit path/to/new_or_modified/file.cpp -m'Commit message'``  #Validate a change. It can be done several ti
git push  #Push your changes into GitHub. Equivalent of "bzr commit", except that your are commiting to your ow
```

Changes will be pushed to your personal "fork", If you have tested your changes and you are ready to push them into the main trunk, just do a "pull request" [5] or create a patch from your commit via:

```
git format-patch origin  #create patch file in current folder)
```

and send to the developers maililng list (yade-dev@lists.launchpad.net) as attachment. In either way, after reviewing your changes they will be added to the main trunk.

When the pull request has been reviewed and accepted, your changes are integrated in the main trunk. Everyone will get them via `git fetch`.

Updating

You may want to get changes done by others:

```
git fetch upstream  #Pull new updates from the upstream to your branch. Eq. of "bzr update", updating the remot
git merge upstream/master  #Merge upstream changes into your master-branch (eq. of "bzr update", updating your
```

Alternatively, this will do fetch+merge all at once (discouraged if you have uncommited changes):

```
git pull
```

7.3 Working directly on git/trunk (recommended for frequent commits)

This direct access to trunk will sound more familiar to bzr or svn users. It is only possible for members of the git team "developpers". Send an email at yade-dev@lists.launchpad.net to join this team (don't forget to tell your git account name).

- Get trunk:

  ```
  git clone git@github.com:yade/trunk.git
  ```

  This creates a new folder, named trunk, that contains the whole code.

- Update

  ```
  git pull
  ```

- Commit to local repository

  ```
  git commit filename1 filename2 ...
  ```

- Push changes to remote trunk

  ```
  git push
  ```

  Now, the changes you made are included in the on-line code, and can be get back by every user.

To avoid confusing logs after each commit/pull/push cycle, it is better to setup automatic rebase:

```
git config --global branch.autosetuprebase always
```

Now your file ~/.gitconfig should include:

[**branch**] autosetuprebase = always

Check also .git/config file in your local trunk folder (rebase = true):

[**branch "master"**] remote = origin

merge = refs/heads/master

rebase = true

Auto-rebase may have unpleasant side effects by blocking "pull" if you have uncommited changes. In this case you can use "git stash":

```
git pull
lib/SConscript: needs update
refusing to pull with rebase: your working tree is not up-to-date
git stash #hide the uncommited changes away
git pull  #now it's ok
git push  #push the commited changes
git stash pop #get uncommited changes back
```

7.4 General guidelines for pushing to yade/trunk

1. Set autorebase once on the computer! (see above)
2. Inspect the diff to make sure you will not commit junk code (typically some "cout<<" left here and there), using in terminal:

   ```
   git diff file1
   ```

   Or, alternatively, any GUI for git: gitg, git-cola...

3. Commit selectively:

```
git commit file1 file2 file3 -m "message" # is good
git commit -a -m "message"                # is bad. It is the best way to commit things that should not be
```

4. Be sure to work with an up-to-date version launching:

```
git pull
```

5. Make sure it compiles and that regression tests pass: try "yade –test" and "yade –check".
6. You can finally let all Yade-users enjoy your work:

```
git push
```

Thanks a lot for your cooperation to Yade!

Chapter 8

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.

8.1 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/.

8.2 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.

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/)

[Bance2014] Bance, S., Fischbacher, J., Schrefl, T., Zins, I., Rieger, G., Cassignol, C. (2014), (Micromagnetics of shape anisotropy based permanent magnets). *Journal of Magnetism and Magnetic Materials* (363), pages 121–124.

[Bonilla2015] Bonilla-Sierra, V., Scholtès, L., Donzé, F.V., Elmouttie, M.K. (2015), (Rock slope stability analysis using photogrammetric data and dfn–dem modelling). *Acta Geotechnica*, pages 1-15. DOI 10.1007/s11440-015-0374-z (fulltext)

[Boon2012a] Boon,C.W., Houlsby, G.T., Utili, S. (2012), (A new algorithm for contact detection between convex polygonal and polyhedral particles in the discrete element method). *Computers and Geotechnics* (44), pages 73 - 82. DOI 10.1016/j.compgeo.2012.03.012 (fulltext)

[Boon2012b] Boon,C.W., Houlsby, G.T., Utili, S. (2013), (A new contact detection algorithm for three-dimensional non-spherical particles). *Powder Technology*. DOI 10.1016/j.powtec.2012.12.040 (fulltext)

[Boon2014] Boon, C.W., Houlsby, G.T., Utili, S. (2014), (New insights into the 1963 vajont slide using 2d and 3d distinct-element method analyses). *Géotechnique* (64), pages 800–816.

[Boon2015] Boon, C.W., Houlsby, G.T., Utili, S. (2015), (A new rock slicing method based on linear programming). *Computers and Geotechnics* (65), pages 12–29.

[Bourrier2013] Bourrier, F., Kneib, F., Chareyre, B., Fourcaud, T. (2013), (Discrete modeling of granular soils reinforcement by plant roots). *Ecological Engineering*. DOI 10.1016/j.ecoleng.2013.05.002 (fulltext)

[Bourrier2015] Bourrier, F., Lambert, S., Baroth, J. (2015), (A reliability-based approach for the design of rockfall protection fences). *Rock Mechanics and Rock Engineering* (48), pages 247–259.

[Catalano2014a] Catalano, E., Chareyre, B., Barthélémy, E. (2014), (Pore-scale modeling of fluid-particles interaction and emerging poromechanical effects). *International Journal for Numerical and Analytical Methods in Geomechanics* (38), pages 51–71. DOI 10.1002/nag.2198 (fulltext) (http://arxiv.org/pdf/1304.4895.pdf)

[Chareyre2012a] Chareyre, B., Cortis, A., Catalano, E., Barthélemy, E. (2012), (Pore-scale modeling of viscous flow and induced forces in dense sphere packings). *Transport in Porous Media* (92), pages 473-493. DOI 10.1007/s11242-011-9915-6 (fulltext)

[Chen2007] Chen, F., Drumm, E. C., Guiochon, G. (2007), (Prediction/verification of particle motion in one dimension with the discrete-element method). *International Journal of Geomechanics, ASCE* (7), pages 344–352. DOI 10.1061/(ASCE)1532-3641(2007)7:5(344)

[Chen2011a] Chen, F., Drumm, E., Guiochon G. (2011), (Coupled discrete element and finite volume solution of two classical soil mechanics problems). *Computers and Geotechnics*. DOI 10.1016/j.compgeo.2011.03.009 (fulltext)

[Chen2012] Chen, Jingsong, Huang, Baoshan, Chen, Feng, Shu, Xiang (2012), (Application of discrete element method to superpave gyratory compaction). *Road Materials and Pavement Design* (13), pages 480-500. DOI 10.1080/14680629.2012.694160 (fulltext)

[Chen2014] Chen, J., Huang, B., Shu, X., Hu, C. (2014), (Dem simulation of laboratory compaction of asphalt mixtures using an open source code). *Journal of Materials in Civil Engineering*.

[Dang2010a] Dang, H. K., Meguid, M. A. (2010), (Algorithm to generate a discrete element specimen with predefined properties). *International Journal of Geomechanics* (10), pages 85-91. DOI 10.1061/(ASCE)GM.1943-5622.0000028

[Dang2010b] Dang, H. K., Meguid, M. A. (2010), (Evaluating the performance of an explicit dynamic relaxation technique in analyzing non-linear geotechnical engineering problems). *Computers and Geotechnics* (37), pages 125 - 131. DOI 10.1016/j.compgeo.2009.08.004

[Donze2008] Donzé, F.V. (2008), (Impacts on cohesive frictional geomaterials). *European Journal of Environmental and Civil Engineering* (12), pages 967–985.

[Duriez2011] Duriez,J., Darve, F., Donzé, F.V. (2011), (A discrete modeling-based constitutive relation for infilled rock joints). *International Journal of Rock Mechanics & Mining Sciences* (48), pages 458–468. DOI 10.1016/j.ijrmms.2010.09.008

[Duriez2013] Duriez, J., Darve, F., Donzé, F.V. (2013), (Incrementally non-linear plasticity applied to rock joint modelling). *International Journal for Numerical and Analytical Methods in Geomechanics* (37), pages 453–477. DOI 10.1002/nag.1105 (fulltext)

[Dyck2015] Dyck, N. J, Straatman, A.G. (2015), (A new approach to digital generation of spherical void phase porous media microstructures). *International Journal of Heat and Mass Transfer* (81), pages 470–477.

[Elias2014] Jan Elias (2014), (Simulation of railway ballast using crushable polyhedral particles). *Powder Technology* (264), pages 458–465. DOI 10.1016/j.powtec.2014.05.052

[Epifancev2013] Epifancev, K., Nikulin, A., Kovshov, S., Mozer, S., Brigadnov, I. (2013), (Modeling of peat mass process formation based on 3d analysis of the screw machine by the code yade). *American Journal of Mechanical Engineering* (1), pages 73–75. DOI 10.12691/ajme-1-3-3 (fulltext)

[Epifantsev2012] Epifantsev, K., Mikhailov, A., Gladky, A. (2012), (Proizvodstvo kuskovogo torfa, ekstrudirovanie, forma zakhodnoi i kalibriruyushchei chasti fil'ery matritsy, metod diskretnykh elementov [rus]). *Mining informational and analytical bulletin (scientific and technical journal)*, pages 212-219.

[Favier2009a] Favier, L., Daudon, D., Donzé, F.V., Mazars, J. (2009), (Predicting the drag coefficient of a granular flow using the discrete element method). *Journal of Statistical Mechanics: Theory and Experiment* (2009), pages P06012.

[Favier2012] Favier, L., Daudon, D., Donzé, F.V. (2012), (Rigid obstacle impacted by a supercritical cohesive granular flow using a 3d discrete element model). *Cold Regions Science and Technology* (85), pages 232–241. (fulltext)

[Gladky2014] Gladkyy, Anton, Schwarze, Rüdiger (2014), (Comparison of different capillary bridge models for application in the discrete element method). *Granular Matter*, pages 1-10. DOI 10.1007/s10035-014-0527-z (fulltext)

[Grujicic2013] Grujicic, M, Snipes, JS, Ramaswami, S, Yavari, R (2013), (Discrete element modeling and analysis of structural collapse/survivability of a building subjected to improvised explosive device (ied) attack). *Advances in Materials Science and Applications* (2), pages 9–24.

[Guo2014] Guo, Ning, Zhao, Jidong (2014), (A coupled fem/dem approach for hierarchical multiscale modelling of granular media). *International Journal for Numerical Methods in Engineering* (99), pages 789–818. DOI 10.1002/nme.4702 (fulltext)

[Guo2015] N. Guo, J. Zhao (2015), (Multiscale insights into classical geomechanics problems). *International Journal for Numerical and Analytical Methods in Geomechanics*. (under review)

[Gusenbauer2012] Gusenbauer, M., Kovacs, A., Reichel, F., Exl, L., Bance, S., Özelt, H., Schrefl, T. (2012), (Self-organizing magnetic beads for biomedical applications). *Journal of Magnetism and Magnetic Materials* (324), pages 977–982.

[Gusenbauer2014] Gusenbauer, M., Nguyen, H., Reichel, F., Exl, L., Bance, S., Fischbacher, J., Özelt, H., Kovacs, A., Brandl, M., Schrefl, T. (2014), (Guided self-assembly of magnetic beads for biomedical applications). *Physica B: Condensed Matter* (435), pages 21–24.

[Hadda2013] Hadda, Nejib, Nicot, François, Bourrier, Franck, Sibille, Luc, Radjai, Farhang, Darve, Félix (2013), (Micromechanical analysis of second order work in granular media). *Granular Matter* (15), pages 221–235. DOI 10.1007/s10035-013-0402-3 (fulltext)

[Hadda2015] Hadda, N., Nicot, F., Wan, R., Darve, F. (2015), (Microstructural self-organization in granular materials during failure). *Comptes Rendus Mécanique*.

[Harthong2009] Harthong, B., Jerier, J.F., Doremus, P., Imbault, D., Donzé, F.V. (2009), (Modeling of high-density compaction of granular materials by the discrete element method). *International Journal of Solids and Structures* (46), pages 3357–3364. DOI 10.1016/j.ijsolstr.2009.05.008

[Harthong2012b] Harthong, B., Jerier, J.-F., Richefeu, V., Chareyre, B., Doremus, P., Imbault, D., Donzé, F.V. (2012), (Contact impingement in packings of elastic–plastic spheres, application to powder compaction). *International Journal of Mechanical Sciences* (61), pages 32–43.

[Hartong2012a] Harthong, B., Scholtès, L., Donzé, F.-V. (2012), (Strength characterization of rock masses, using a coupled dem–dfn model). *Geophysical Journal International* (191), pages 467–480. DOI 10.1111/j.1365-246X.2012.05642.x (fulltext)

[Hassan2010] Hassan, A., Chareyre, B., Darve, F., Meyssonier, J., Flin, F. (2010 (submitted)), (Microtomography-based discrete element modelling of creep in snow). *Granular Matter*.

[Hilton2013] Hilton, J. E., Tordesillas, A. (2013), (Drag force on a spherical intruder in a granular bed at low froude number). *Phys. Rev. E* (88), pages 062203. DOI 10.1103/PhysRevE.88.062203 (fulltext)

[Jerier2009] Jerier, J.-F., Imbault, D.and Donzé, F.V., Doremus, P. (2009), (A geometric algorithm based on tetrahedral meshes to generate a dense polydisperse sphere packing). *Granular Matter* (11). DOI 10.1007/s10035-008-0116-0

[Jerier2010a] Jerier, J.-F., Richefeu, V., Imbault, D., Donzé, F.V. (2010), (Packing spherical discrete elements for large scale simulations). *Computer Methods in Applied Mechanics and Engineering*. DOI 10.1016/j.cma.2010.01.016

[Jerier2010b] Jerier, J.-F., Hathong, B., Richefeu, V., Chareyre, B., Imbault, D., Donzé, F.-V., Doremus, P. (2010), (Study of cold powder compaction by using the discrete element method). *Powder Technology* (In Press). DOI 10.1016/j.powtec.2010.08.056

[Kozicki2006a] Kozicki, J., Tejchman, J. (2006), (2D lattice model for fracture in brittle materials). *Archives of Hydro-Engineering and Environmental Mechanics* (53), pages 71–88. (fulltext)

[Kozicki2007a] Kozicki, J., Tejchman, J. (2007), (Effect of aggregate structure on fracture process in concrete using 2d lattice model"). *Archives of Mechanics* (59), pages 365–384. (fulltext)

[Kozicki2008] Kozicki, J., Donzé, F.V. (2008), (A new open-source software developed for numerical simulations using discrete modeling methods). *Computer Methods in Applied Mechanics and Engineering* (197), pages 4429–4443. DOI 10.1016/j.cma.2008.05.023 (fulltext)

[Kozicki2009] Kozicki, J., Donzé, F.V. (2009), (Yade-open dem: an open-source software using a discrete element method to simulate granular material). *Engineering Computations* (26), pages 786–805. DOI 10.1108/02644400910985170 (fulltext)

[Lomine2013] Lominé, F., Scholtès, L., Sibille, L., Poullain, P. (2013), (Modelling of fluid-solid interaction in granular media with coupled lb/de methods: application to piping erosion). *Interna-*

tional Journal for Numerical and Analytical Methods in Geomechanics (37), pages 577-596. 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

[Nicot2013a] Nicot, F., Hadda, N., Darve, F. (2013), (Second-order work analysis for granular materials using a multiscale approach). *International Journal for Numerical and Analytical Methods in Geomechanics*. DOI 10.1002/nag.2175

[Nicot2013b] Nicot, F., Hadda, N., Guessasma, M., Fortin, J., Millet, O. (2013), (On the definition of the stress tensor in granular media). *International Journal of Solids and Structures*. DOI 10.1016/j.ijsolstr.2013.04.001 (fulltext)

[Nitka2015] Nitka, M., Tejchman, J. (2015), (Modelling of concrete behaviour in uniaxial compression and tension with dem). *Granular Matter*, pages 1–20.

[Puckett2011] Puckett, J.G., Lechenault, F., Daniels, K.E. (2011), (Local origins of volume fraction fluctuations in dense granular materials). *Physical Review E* (83), pages 041301. DOI 10.1103/PhysRevE.83.041301 (fulltext)

[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 (fulltext)

[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 (fulltext)

[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. (fulltext)

[Scholtes2010] Scholtès, L., Hicher, P.-Y., Sibille, L. (2010), (Multiscale approaches to describe mechanical responses induced by particle removal in granular materials). *Comptes Rendus Mécanique* (338), pages 627–638. DOI 10.1016/j.crme.2010.10.003 (fulltext)

[Scholtes2011] Scholtès, L., Donzé, F.V., Khanal, M. (2011), (Scale effects on strength of geomaterials, case study: coal). *Journal of the Mechanics and Physics of Solids* (59), pages 1131–1146. DOI 10.1016/j.jmps.2011.01.009 (fulltext)

[Scholtes2012] Scholtès, L., Donzé, F.V. (2012), (Modelling progressive failure in fractured rock masses using a 3d discrete element method). *International Journal of Rock Mechanics and Mining Sciences* (52), pages 18–30. DOI 10.1016/j.ijrmms.2012.02.009 (fulltext)

[Scholtes2013] Scholtès, L., Donzé, F.V. (2013), (A DEM model for soft and hard rocks: role of grain interlocking on strength). *Journal of the Mechanics and Physics of Solids* (61), pages 352–369. DOI 10.1016/j.jmps.2012.10.005 (fulltext)

[Scholtes2015a] Scholtès, L., Chareyre, B., Michallet, H., Catalano, E., Marzougui, D. (2015), (Modeling wave-induced pore pressure and effective stress in a granular seabed). *Continuum Mechanics and Thermodynamics* (27), pages 305–323. DOI http://dx.doi.org/10.1007/s00161-014-0377-2

[Scholtes2015b] Scholtès, L., Donzé, F., V. (2015), (A dem analysis of step-path failure in jointed rock slopes). *Comptes rendus - Mécanique* (343), pages 155–165. DOI http://dx.doi.org/10.1016/j.crme.2014.11.002

[Shiu2008] Shiu, W., Donzé, F.V., Daudeville, L. (2008), (Compaction process in concrete during missile impact: a dem analysis). *Computers and Concrete* (5), pages 329–342.

[Shiu2009] Shiu, W., Donzé, F.V., Daudeville, L. (2009), (Discrete element modelling of missile impacts on a reinforced concrete target). *International Journal of Computer Applications in Technology* (34), pages 33–41.

[Sibille2014] Sibille, L., Lominé, F., Poullain, P., Sail, Y., Marot, D. (2014), (Internal erosion in granular media: direct numerical simulations and energy interpretation). *Hydrological Processes*. DOI 10.1002/hyp.10351 (fulltext) (First published online Oct. 2014)

[Sibille2015] Sibille, L., Hadda, N., Nicot, F., Tordesillas, A., Darve, F. (2015), (Granular plasticity, a contribution from discrete mechanics). *Journal of the Mechanics and Physics of Solids* (75), pages 119–139. DOI 10.1016/j.jmps.2014.09.010 (fulltext)

[Smilauer2006] Václav Šmilauer (2006), (The splendors and miseries of yade design). *Annual Report of Discrete Element Group for Hazard Mitigation*. (fulltext)

[Thoeni2013] K. Thoeni, C. Lambert, A. Giacomini, S.W. Sloan (2013), (Discrete modelling of hexagonal wire meshes with a stochastically distorted contact model). *Computers and Geotechnics* (49), pages 158–169. DOI 10.1016/j.compgeo.2012.10.014 (fulltext)

[Thoeni2014] K. Thoeni, A. Giacomini, C. Lambert, S.W. Sloan, J.P. Carter (2014), (A 3D discrete element modelling approach for rockfall analysis with drapery systems). *International Journal of Rock Mechanics and Mining Sciences* (68), pages 107–119. DOI 10.1016/j.ijrmms.2014.02.008 (fulltext)

[Tong2012] Tong, A.-T., Catalano, E., Chareyre, B. (2012), (Pore-scale flow simulations: model predictions compared with experiments on bi-dispersed granular assemblies). *Oil & Gas Science and Technology - Rev. IFP Energies nouvelles*. DOI 10.2516/ogst/2012032 (fulltext)

[Tran2011] Tran, V.T., Donzé, F.V., Marin, P. (2011), (A discrete element model of concrete under high triaxial loading). *Cement and Concrete Composites*.

[Tran2012] Tran, V.D.H., Meguid, M.A., Chouinard, L.E. (2012), (An algorithm for the propagation of uncertainty in soils using the discrete element method). *The Electronic Journal of Geotechnical Engineering*. (fulltext)

[Tran2012c] Tran, V.D.H., Meguid, M.A., Chouinard, L.E. (2012), (Discrete element and experimental investigations of the earth pressure distribution on cylindrical shafts). *International Journal of Geomechanics*. DOI 10.1061/(ASCE)GM.1943-5622.0000277

[Tran2013] Tran, V.D.H., Meguid, M.A., Chouinard, L.E. (2013), (A finite–discrete element framework for the 3d modeling of geogrid–soil interaction under pullout loading conditions). *Geotextiles and Geomembranes* (37), pages 1-9. DOI 10.1016/j.geotexmem.2013.01.003

[Tran2014] Tran, VDH, Meguid, MA, Chouinard, LE (2014), (Three-dimensional analysis of geogrid-reinforced soil using a finite-discrete element framework). *International Journal of Geomechanics*.

[Wan2014] Wan, R, Khosravani, S, Pouragha, M (2014), (Micromechanical analysis of force transport in wet granular soils). *Vadose Zone Journal* (13).

[Wang2014] Wang, XiaoLiang, Li, JiaChun (2014), (Simulation of triaxial response of granular materials by modified dem). *Science China Physics, Mechanics & Astronomy* (57), pages 2297–2308.

[Zhao2015] J. Zhao, N. Guo (2015), (The interplay between anisotropy and strain localisation in granular soils: a multiscale insight). *Géotechnique*. (under review)

[kozicki2014] Kozicki, Jan, Tejchman, Jacek, Mühlhaus, Hans-Bernd (2014), (Discrete simulations of a triaxial compression test for sand by dem). *International Journal for Numerical and Analytical Methods in Geomechanics* (38), pages 1923–1952.

[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)

[Catalano2012] Emanuele Catalano (2012), (A pore-scale coupled hydromechanical model for biphasic granular media). PhD thesis at *Université de Grenoble*. (fulltext)

[Charlas2013] Benoit Charlas (2013), (Etude du comportement mécanique d'un hydrure intermétallique utilisé pour le stockage d'hydrogène). PhD thesis at *Université de 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)

[Chen2011b] Chen, J. (2011), (Discrete element method (dem) analyses for hot-mix asphalt (hma) mixture compaction). PhD thesis at *University of Tennessee, Knoxville*. (fulltext)

[Duriez2009a] J. Duriez (2009), (Stabilité des massifs rocheux : une approche mécanique). PhD thesis at *Institut polytechnique de Grenoble*. (fulltext)

[Favier2009c] Favier, L. (2009), (Approche numérique par éléments discrets 3d de la sollicitation d'un écoulement granulaire sur un obstacle). PhD thesis at *Université Grenoble I – Joseph Fourier*.

[Guo2014c] N. Guo (2014), (Multiscale characterization of the shear behavior of granular media). PhD thesis at *The Hong Kong University of Science and Technology*.

[Jerier2009b] Jerier, J.F. (2009), (Modélisation de la compression haute densité des poudres métalliques ductiles par la méthode des éléments discrets (in french)). PhD thesis at *Université Grenoble I – Joseph Fourier*. (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)

[Tran2011b] 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)

[Antypov2011] D. Antypov, J. A. Elliott (2011), (On an analytical solution for the damped hertzian spring). *EPL (Europhysics Letters)* (94), pages 50004. (fulltext)

[Bagi2006] Katalin Bagi (2006), (Analysis of microstructural strain tensors for granular assemblies). *International Journal of Solids and Structures* (43), pages 3166 - 3184. DOI 10.1016/j.ijsolstr.2005.07.016

[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.

[Calvetti1997] Calvetti, F., Combe, G., Lanier, J. (1997), (Experimental micromechanical analysis of a 2d granular material: relation between structure evolution and loading path). *Mechanics of Cohesive-frictional Materials* (2), pages 121–163.

[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.

[Chan2011] D. Chan, E. Klaseboer, R. Manica (2011), (Film drainage and coalescence between deformable drops and bubbles.). *Soft Matter* (7), pages 2235-2264.

[Chareyre2002a] B. Chareyre, L. Briancon, P. Villard (2002), (Theoretical versus experimental modeling of the anchorage capacity of geotextiles in trenches.). *Geosynthet. Int.* (9), pages 97–123.

[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

[Dallavalle1948] J. M. DallaValle (1948), (Micrometrics : the technology of fine particles). Pitman Pub. Corp.

[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.

[GarciaRojo2004] R. García-Rojo, S. McNamara, H. J. Herrmann (2004), (Discrete element methods for the micro-mechanical investigation of granular ratcheting). In *Proceedings ECCOMAS 2004*. (fulltext)

[Hentz2003] Séebastien Hentz (2003), (Modélisation d'une structure en béton armé soumise à un choc par la méthode des eléments discrets). PhD thesis at *Université Grenoble 1 – Joseph Fourier*.

[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.

[Herminghaus2005] Herminghaus, S. (2005), (Dynamics of wet granular matter). *Advances in Physics* (54), pages 221-261. DOI 10.1080/00018730500167855 (fulltext)

[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

[Ivars2011] Diego Mas Ivars, Matthew E. Pierce, Caroline Darcel, Juan Reyes-Montes, David O. Potyondy, R. Paul Young, Peter A. Cundall (2011), (The synthetic rock mass approach for jointed rock mass modelling). *International Journal of Rock Mechanics and Mining Sciences* (48), pages 219 - 244. DOI 10.1016/j.ijrmms.2010.11.014

[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)

[Lambert2008] Lambert, Pierre, Chau, Alexandre, Delchambre, Alain, Régnier, Stéphane (2008), (Comparison between two capillary forces models). *Langmuir* (24), pages 3157–3163.

[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)

[Lucy1977] Lucy, L.~B. (1977), (A numerical approach to the testing of the fission hypothesis). *aj* (82), pages 1013-1024. DOI 10.1086/112164 (fulltext)

[Luding2008] Stefan Luding (2008), (Introduction to discrete element methods). In *European Journal of Environmental and Civil Engineering*.

[Luding2008b] Luding, Stefan (2008), (Cohesive, frictional powders: contact models for tension). *Granular Matter* (10), pages 235-246. DOI 10.1007/s10035-008-0099-x (fulltext)

[Magnier1998a] S.A. Magnier, F.V. Donzé (1998), (Numerical simulation of impacts using a discrete element method). *Mech. Cohes.-fric. Mater.* (3), pages 257–276. DOI 10.1002/(SICI)1099-1484(199807)3:3<257::AID-CFM50>3.0.CO;2-Z

[Mani2013] Mani, Roman, Kadau, Dirk, Herrmann, HansJ. (2013), (Liquid migration in sheared unsaturated granular media). *Granular Matter* (15), pages 447-454. DOI 10.1007/s10035-012-0387-3 (fulltext)

[McNamara2008] S. McNamara, R. García-Rojo, H. J. Herrmann (2008), (Microscopic origin of granular ratcheting). *Physical Review E* (77). DOI 11.1103/PhysRevE.77.031304

[Monaghan1985] Monaghan, J.~J., Lattanzio, J.~C. (1985), (A refined particle method for astrophysical problems). *aap* (149), pages 135-143. (fulltext)

[Monaghan1992] Monaghan, J.~J. (1992), (Smoothed particle hydrodynamics). *araa* (30), pages 543-574. DOI 10.1146/annurev.aa.30.090192.002551

[Morris1997] (1997), (Modeling low reynolds number incompressible flows using {sph}). *Journal of Computational Physics* (136), pages 214 - 226. DOI http://dx.doi.org/10.1006/jcph.1997.5776 (fulltext) ()

[Mueller2003] Müller, Matthias, Charypar, David, Gross, Markus (2003), (Particle-based fluid simulation for interactive applications). In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. (fulltext)

[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)

[Potyondy2004] D.O. Potyondy, P.A. Cundall (2004), (A bonded-particle model for rock). *International Journal of Rock Mechanics and Mining Sciences* (41), pages 1329 - 1364. DOI 10.1016/j.ijrmms.2004.09.011

[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)

[Rabinov2005] RABINOVICH Yakov I., ESAYANUR Madhavan S., MOUDGIL Brij M. (2005), (Capillary forces between two spheres with a fixed volume liquid bridge : theory and experiment). *Langmuir* (21), pages 10992–10997. (fulltext) (eng)

[Radjai2011] Radjai, F., Dubois, F. (2011), (Discrete-element modeling of granular materials). John Wiley & Sons. (fulltext)

[RevilBaudard2013] Revil-Baudard, T., Chauchat, J. (2013), (A two-phase model for sheet flow regime based on dense granular flow rheology). *Journal of Geophysical Research: Oceans* (118), pages 619–634.

[Richarson1954] Richardson, J. F.„ W. N. Zaki (1954), (Sedimentation and fluidization: part i). *Trans. Instn. Chem. Engrs* (32).

[Satake1982] M. Satake (1982), (Fabric tensor in granular materials.). In *Proc., IUTAM Symp. on Deformation and Failure of Granular materials, Delft, The Netherlands.*

[Schmeeckle2007] Schmeeckle, Mark W., Nelson, Jonathan M., Shreve, Ronald L. (2007), (Forces on stationary particles in near-bed turbulent flows). *Journal of Geophysical Research: Earth Surface* (112). DOI 10.1029/2006JF000536 (fulltext)

[Schwager2007] Schwager, Thomas, Pöschel, Thorsten (2007), (Coefficient of restitution and linear–dashpot model revisited). *Granular Matter* (9), pages 465-469. DOI 10.1007/s10035-007-0065-z (fulltext)

[Soulie2006] Soulié, F., Cherblanc, F., El Youssoufi, M.S., Saix, C. (2006), (Influence of liquid bridges on the mechanical behaviour of polydisperse granular materials). *International Journal for Numerical and Analytical Methods in Geomechanics* (30), pages 213–228. DOI 10.1002/nag.476 (fulltext)

[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)

[Weigert1999] Weigert, Tom, Ripperger, Siegfried (1999), (Calculation of the liquid bridge volume and bulk saturation from the half-filling angle). *Particle & Particle Systems Characterization* (16), pages 238–242. DOI 10.1002/(SICI)1521-4117(199910)16:5<238::AID-PPSC238>3.0.CO;2-E (fulltext)

[Wiberg1985] Wiberg, Patricia L., Smith, J. Dungan (1985), (A theoretical model for saltating grains in water). *Journal of Geophysical Research: Oceans* (90), pages 7341–7354.

[Willett2000] Willett, Christopher D., Adams, Michael J., Johnson, Simon A., Seville, Jonathan P. K. (2000), (Capillary bridges between two spherical bodies). *Langmuir* (16), pages 9396-9405. DOI 10.1021/la000657y (fulltext)

[Zhou1999536] Y.C. Zhou, B.D. Wright, R.Y. Yang, B.H. Xu, A.B. Yu (1999), (Rolling friction in the dynamic simulation of sandpile formation). *Physica A: Statistical Mechanics and its Applications* (269), pages 536–553. DOI 10.1016/S0378-4371(99)00183-1 (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.