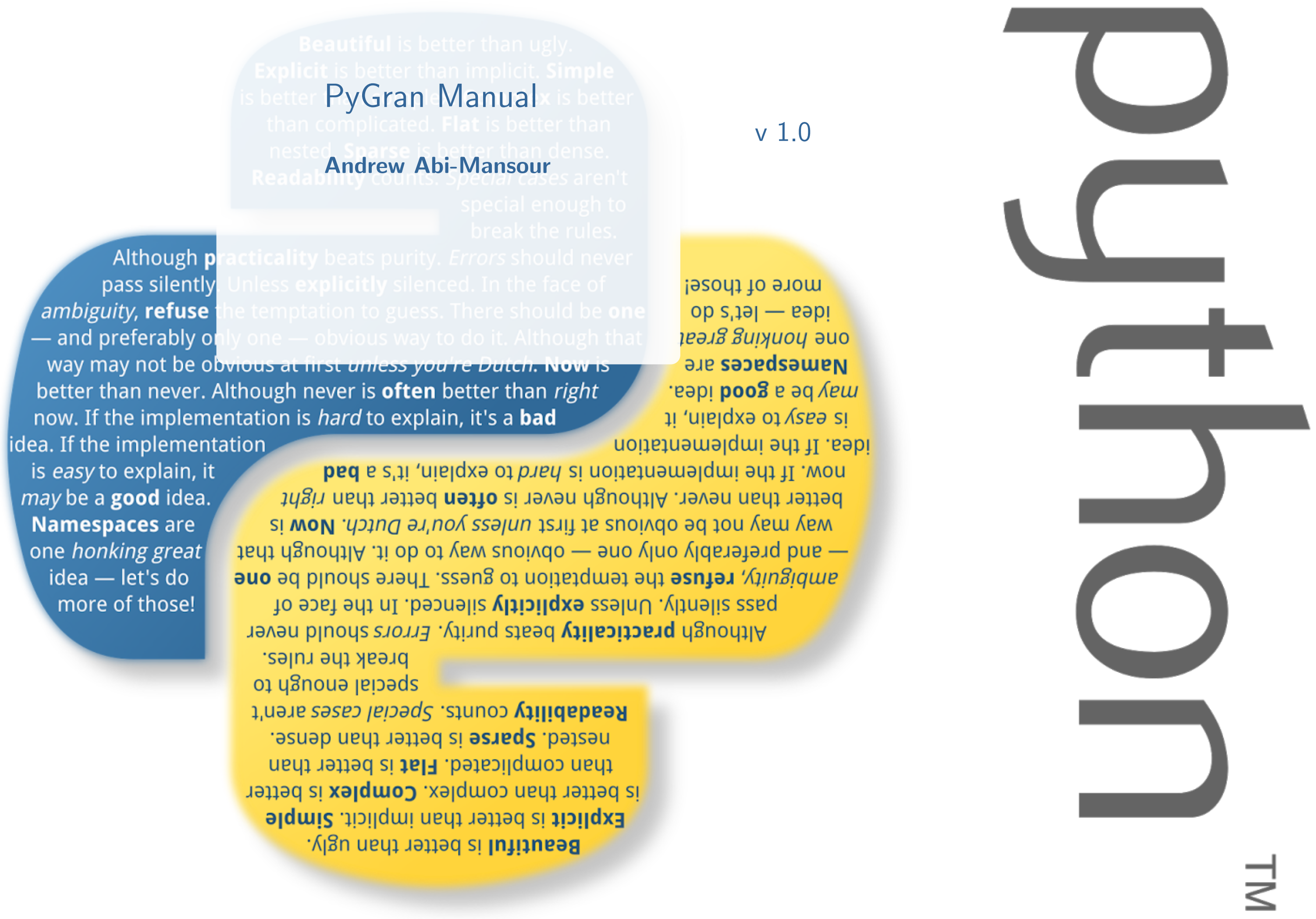
PyGran Manual

v 1.0

Andrew Abi-Mansour



Contents

Part I

Preliminary

Chapter 1

Introduction

PyGran is an object-oriented library written primarily in Python for DEM simulation and analysis. The main purpose of *PyGran* is to provide an easy and intuitive way for performing technical computing in DEM, enabling flexibility in how users interact with data from the onset of a simulation and until the post-processing stage. In addition to providing a brief tutorial on installing *PyGran* for Unix systems (Part I), this manual focuses on two core modules (Figure (1.1)) in *PyGran*: *Simulator* (Part II) which provides *engines* for running DEM simulations and enables analysis of contact mechanical models, and *Analyzer* (Part III) which contains methods and modules for processing DEM data. *PyGran* is released under the GNU General Public License (GPL v2.0), and its code base is available from github.

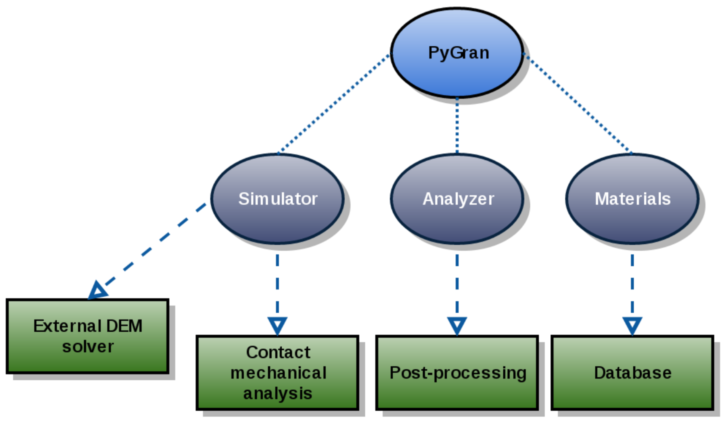


Figure 1.1: A diagram that shows the hierarchical structure of *PyGran* in terms of its core modules and submodules that can be imported from a Python input script.

Chapter 2

Prerequisites

2.1 OS support

In the current version (1.0), *PyGran* is configured to run on Unix or Unix-like operating systems. While *PyGran* can be run on a Windows platform, it has not yet been fully tested. *PyGran* supports Python 3.X and is fully backwards compatible with Python 2.2 (and later verions). Table (2.1) summarizes some of the technical details of the *PyGran* source code.

2.2 Dependencies

2.2.1 Core packages

PyGran is designed to work in conjunction with NumPy, SciPy, and other Python libraries. The following packages must be installed before *PyGran* is configured to run. For running DEM simluations with *LIGGGHTS* [5] in parallel, OpenMPI or MPICH2 must be installed on the system. Furthermore, the following four Python packages must be installed:

Nr.	**Code metadata description**	
C1	Current code version	v1.0
C2	Permanent link to code/repository used for this code version	https://github.com/PyGran
C3	Legal Code License	GNU General Public License v2.0
C4	Code versioning system used	Git
C5	Software code languages, tools, and services used	Python, Cython, MPI4Py
C6	Compilation requirements, operating environments	OS: Linux, Mac OS X; C compiler: gcc
C7	If available Link to developer documentation/manual	
C8	Developer email address	andrew.gaam@gmail.com

Table 2.1: Code metadata

- *Numpy* : for exposing trajectory data as *ndarray* objects and performing linear algebra floating-point operations
- *Scipy* : for efficient nearest neighbor searching routines, sorting, and non-linear solvers.
- *Cython* : for improved performance in *Analyzer.core* module
- *mpi4py* : for running DEM simulations in parallel with MPI

2.2.2 Optional packages

These packages are needed to achieve full functionality in *PyGran*.

- *PyVTK* : for reading mesh files in VTK file format
- *matplotlib* : for generating 2D plots

For reading input trajectory files in vtk/vtu format, the VTK library must be installed.

All of *PyGran*'s dependencies can be installed with pip (or pip3 for Python 3.x):

```
pip install numpy scipy mpi4py cython vtk PyVTK matplotlib --user
```

2.3 Installation

2.3.1 Experimental version

Even though *git* is not required to install or run *PyGran*, its availability makes it easier and convenient to download the latest version of the *PyGran* source code via

```
git clone https://github.com/PyGran
```

This clones the repository to a directory called ‘PyGran’:

```
cd PyGran
```

For updating an existing repository, *git* can be used to sync the source code with the online repository via

```
git pull origin master
```

Alternatively, one can download the source code as a tar ball (or zip file) from github.com, and then manually extract the files. *PyGran* uses Python’s ‘setup-tools’ to check for and/or download dependencies. For building *PyGran*, run from the ‘PyGran’ directory:

```
python setup.py build
```

For installing *PyGran*, run from the ‘PyGran’ directory:

```
python setup.py install
```

For a comprehensive list of options on running ‘setup.py’, see the doc strings in setuptools.

2.3.2 Stable version

A stable release of *PyGran* can be downloaded from github and then installed using the method described in subsection (2.3.1), or alternatively using pip:

```
pip install PyGran --user
```

Similarly one could use pip3 to install *PyGran* for Python 3.X.

2.3.3 Configuration with *LIGGGHTS*

PyGran has been successfully tested with *LIGGGHTS* [5] v3.7 and v3.8. For running DEM simulations with *LIGGGHTS*, the latter must be compiled as a shared library (shared object on Unix/Linux systems), which *PyGran* will attempt to find. Be default, *PyGran* searches for 'libliggghts.so'. The user can specify the name of the shared object and its location by writing its full path to *PyGran*/.config file. If .config file does not exist, then it must be created. For example, if a *PyGran* installation exists in /home/user/.local/lib/python3.5/site-packages/PyGran-1.0-py3.5-linux-x86_64.egg then inserting the line:

```
library=/home/user/LIGGGHTS-PUBLIC/src/lmp_mpi.so
```

to /home/user/.local/lib/python3.5/site-packages/PyGran-1.0-py3.5-linux-x86_64.egg/ instructs *PyGran* to look for the file lmp_mpi.so in the user's LIGGGHTS-PUBLIC/src directory.

Part II

Simulation

Chapter 3

Numerical Analysis

PyGran provides a convenient way for users to define materials as Python dictionaries in the *Materials* module. For instance, properties of stearic acid shown in Code (3.1) are available in the *Materials* module. This dictionary can then be used for running simulation or performing analysis.

```
stearicAcid = {
    'youngsModulus': 4.15e7,
    'poissonsRatio': 0.25,
    'coefficientFriction': 0.5,
    'coefficientRollingFriction': 0.0,
    'cohesionEnergyDensity': 0.033,
    'coefficientRestitution': 0.9,
    'coefficientRollingViscousDamping': 0.1,
    'yieldPress': 2.2e6,
    'characteristicVelocity': 0.1,
    'density': 997.164
    }
```

Listing 3.1: A Python dictionary that defines material properties of stearic acid can be conveniently used in various *PyGran* modules and routines.

The *PyGran.Simulator.models* module contains classes for 3 contact mechanical models: *SpringDashpot* [1], *HertzMindlin*, and *ThorntonNing* [2]. While these models can be used to run a DEM simulation with *LIGGGHTS*, they also provide a way for investigating numerical aspects of contact models as shown in the next section.

3.1 Contact mechanical models

PyGran.Simulator.models.model is the basic class from which contact models are derived. This class contains methods that are overwritten by a sublcass that implements a specific contact model. The 3 contact models implemented in *PyGran* are: spring-dashpot [ref], Hertz-Mindlin [ref], and Thornton-Ning [ref]. In the next section, it is demonstrated how these models can be used to perform simple numerical experiments.

3.2 Examples

3.2.1 Hertz-Mindlin vs Spring-Dashpot

Code 3.2 shows how *PyGran.Simulator* can be used to compute the force-displacement curves for two different visco-elastic models: spring-dashpot, and Hertz-Mindlin models.

```
import PyGran.Simulator as Sim

# Use the following two viscoelastic models
models = [Sim.models.SpringDashpot, Sim.models.HertzMindlin]

# Define material properties
powderX = {
  'youngsModulus': 1e8,
  'poissonsRatio': 0.25,
    'coefficientRestitution': 0.9,
    'characteristicVelocity': 0.1,
    'density': 997.164,
    'radius': 1e-4
}

for model in models:

  model = model(material=powderX)
  time, soln, force = model.displacement()

  # Extract normal displacement
  deltan = soln[:,0]

  # Ignore negative (attractive) forces
  deltan = deltan[force >= 0]
  force = force[force >= 0]
```

Listing 3.2: A *PyGran* script that uses the *Simulator* module to compute the visco-elastic force between two spheres of reduced radius set to 100 μm.

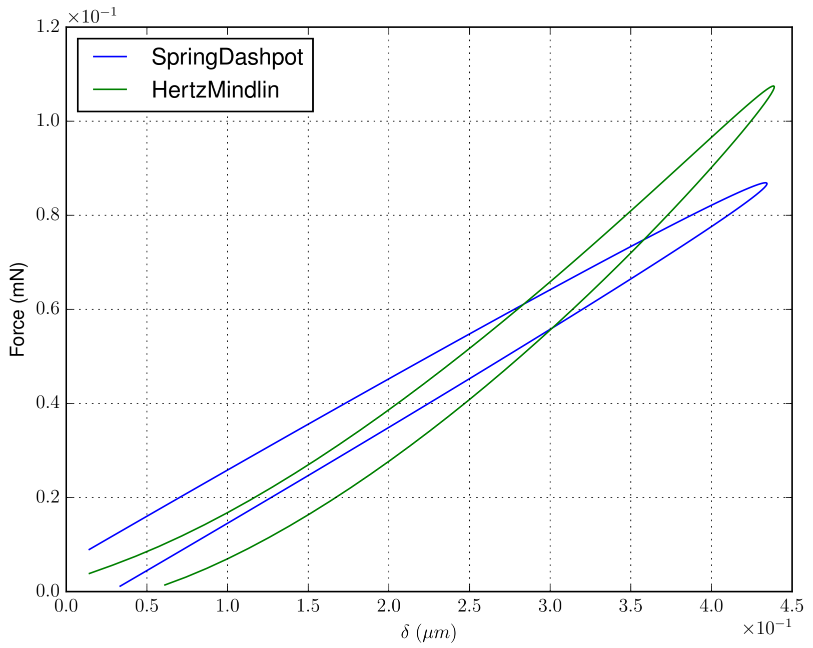


Figure 3.1: Force as a function of displacement (δ) computed for the Spring-Dashpot and Hertz-Mindlin models available in the *Simulator.models* module.

3.2.2 Coefficient of restitution

An elasto-plastic contact model suggested by Thornton and Ning [ref] is available in the *PyGran.Simulator* module.

```
import PyGran.Simulator as Sim
from numpy import arange, fabs

cModel = Sim.models.ThorntonNing

# Define material properties
powderX = {
  'youngsModulus': 1e8,
  'poissonsRatio': 0.25,
  'coefficientRestitution': 0.9,
  'characteristicVelocity': 0.1,
  'density': 997.164,
  'radius': 1e-4
}

# Initialize variables
COR = []
pressure = arange(1e6, 4e6, 1e5)

for yieldPress in pressure:

  powderX['yieldPress'] = yieldPress
  model = cModel(material=powderX)

  time, disp, force = model.displacement()
  deltav = disp[:,1]

  COR.append(fabs(deltav[-1] / deltav[0]))
```

Listing 3.3: A *PyGran* script that uses the *Simulator* module to compute the elasto-plastic force between two spheres of reduced radius set to 100 μm.

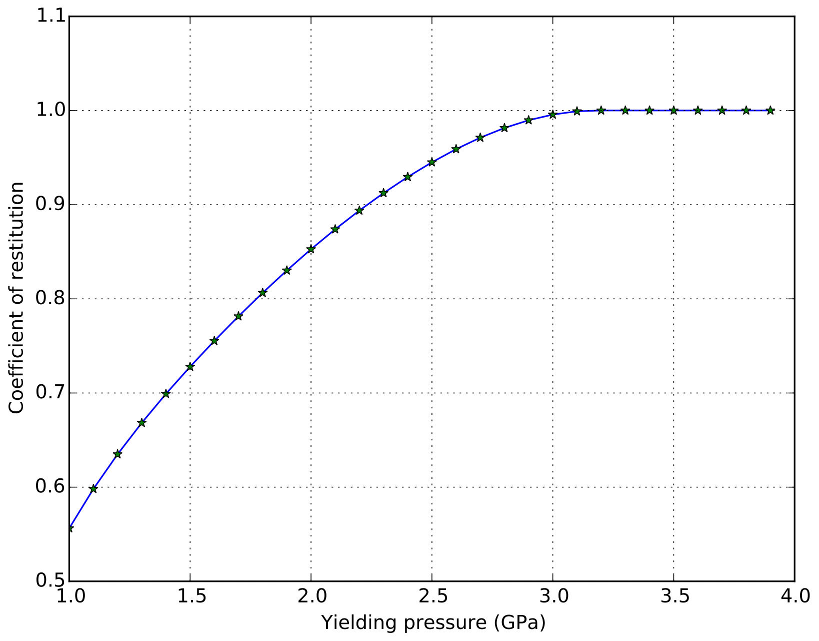


Figure 3.2: The coefficient of restitution for two spheres of reduced radius of 100 μm computed using the *ThorntonNing* model implemented in *PyGran*.

Chapter 4

DEM Simulation

4.1 Engines in *PyGran*

External N-body (DEM) solvers such as *LAMMPS* or *LIGGGHTS* can be called from *PyGran.Simulator* provided there is a supported interface that can import this solver as a separate module (in the form of a shared object). An *engine* provides an interface for *PyGran* to call specific methods in the DEM solver. While *PyGran* provides an *engine* for LIGGGHTS, it can be readily used with solvers such as Yade and ESyS-particle that have their own Python APIs.

4.1.1 Fundamentals

Any *PyGran* engine must provide a *Simulator.DEMPy* class that is instantiated by *Simulator.DEM*. The latter must be created by the user at the onset of any DEM simulation:

```
sim = Simulator.DEM(**args)
```

where *args* is a Python dictionary that contains keywords specific to the *engine* selected to setup and run the DEM simulation. The keyword 'engine' (by default

PyGran.Simulator.engines.liggghts) is used to specify which engine to use for running DEM simulation. Currently, *PyGran* supports only *LIGGGHTS* [5] as a viable engine. For that purpose, *LIGGGHTS* must be compiled as a shared library which *PyGran* will attempt to find (or alternatively the user can specify the location of the shared object in a '.config' file created in the *PyGran* directory and adding the line 'library=path/to/libliggghts.so'.

Other specific keywords for the *LIGGGHTS* engine in *PyGran* are discussed in the next section.

4.2 MPI-based Engine: *LIGGGHTS*

The Python dictionary supplied to *Simulator.DEM* can contain many keywords summarized below. The default value used (if any) for each is shown in square brackets.

- Abstract parameters:
  - model [*SpringDashpot*]: a *Simulator.models.Model* object that specifies the contact mechanical model. *Model* can be *SpringDashpot*, *HertzMindlin*, or *ThorntonNing*
  - nSim [1]: number of concurrent simulations to launch for parametrization studies
- Computational parameters:
  - units ['si']: a string specifying which unit system to use (see *LIGGGHTS* manual)
  - nns_type ['bin']: a number specifying the algorithm used to build the nearest-neighbor list (bin, nsq ,or multi)

  - nns_skin [4 * max_radius]: a number specifying the skin distance (see *LIGGGHTS* manual)

- System parameters:
  - dim [3]: an integer specifying the spatial dimensions of the simulation
  - box: a tuple if size dim * 3 specifying the box size: (x_min, x_max, y_min, y_max, ...)
  - boundary [sim_box]: a tuple of size *dim* containing strings that specify the boundary conditions (e.g. (‘b_x’, ‘b_y’, ‘b_z’) in 3D) where the string can be ‘p’ (periodic), ‘f’ (fixed), ‘s’ (shrink-wrapped), or ‘m’ (bounded shrink wrapped). A boundary string of the form ’pf’ signifies periodic conditions at the lower face and fixed conditions at the upper face. See the *LIGGGHTS* manual for additional information.
  - gravity: a tuple of length $dim + 1$ speciying the direction and magnitude of gravitional acceleration, e.g. (0, 0, -1, 9.81) specifies gravity to act in the negative z direction with magnitude 9.81.

- Species parameters:
  - SS
  - mesh

- Input/output parameters:
  - output [...]: a string specifying the name of the directory to store all simulation data in. By default, it is the name of the contact model used with the day/hour/min/sec time.
  - restart [...]: a tuple of size 5 containing: (freq, dir, fname, resume, lastfile). freq: an integer specifying how often to write a restart file,

dir: string specifying the name of the directory to write the restart files to, fname: string for the restart filename, resume: boolean variable for resuming simulation, lastfile: a string specifying the filename to resume the simulation from, can be 'None' for resuming the simulation from the last written restart file.

  - dump_modify [('append', 'yes')]: a tuple of arguments to pass to *LIGGGHTS* for dumping files. See *LIGGGHTS* manual.
  - traj [...]: a dictionary containing keywords for dump style. Keywords are: 'sel': species (1,2,.. 'all'), 'freq': int, 'dir': str, 'style': 'custom', 'pfile': 'traj.dump', 'mfile': 'mesh*.vtk', 'args': ('id', 'type', 'x', 'y', 'z', 'radius'), 'margs': ('id', 'stress'). *PyGran* by default writes particles as a dump file and nothing for a mesh. To change this behavior, 'pfile' can be None (no output) or a string specifying the output filename, and similarly for 'mfile' specifying the mesh output filename, which must have a vtk/stl/vtm/vtu extension. The *args* and *margs* keywords contain the attributes for particles or mesh(es) depending on the style and file extension specified. See *LIGGGHTS* manual.

4.2.1 Constructors and destructors

- insert(**args): creates particles by insertion. Returns insertion fix ID. This method has the following arguments:
  - species: either an integer (1,2,3,...) specifying the species type to insert, or the keyword 'all' to insert all defined species
  - insert ['by_pack']: insertion mechanism defined by a string that can be: 'by_pack', 'by_rate', or 'by_stream'. See *LIGGGHTS* manual.
  - value: an integer (e.g. number of particles), or a float (e.g. volume or mass fraction), depending on the mechanism of insertion

– mech [‘particles__region’]: a string specifying the mechanism of insertion. If inserting by pack, this argument can be: ‘particles_in_region’, ‘volumefraction_region’, or ‘mass_in_region’. If inserting by stream or rate, this argument can be: ‘particlerate’, ‘nparticles’, ‘mass’, or ‘massrate’)

– vel_type [‘constant’]: a string that defines the initial velocity type: ‘constant’, ‘uniform’ (random number), or ‘Gaussian’.

– vel [(0,0,0)]: a tuple specifying the initial velocity components of all particles. Its size is: *dim* when *vel_type* is constant, *dim* * 2 when *vel_type* is ‘uniform’, or ‘Gaussian’, e.g. (v_x, v_y, v_z, dv_x, dv_y, dv_z) with the last 3 entries specifying the fluctuation amplitude in the velocity components. See *LIGGGHTS* manual.

– insert [‘by_pack’]: a string specifying how particles are inserted (‘by_rate’, ‘by_pack’, or ‘by_stream’). See

– region [‘sim_box’]: a tuple whose 1st element is a string that indictes the region type such as ‘block’, ‘cone’, ‘cylinder’, etc. and the remaining elements specify the region boundaries, e.g. (‘block’, x_min, x_max, y_min, y_max, z_min, z_max) defines a rectangular 3D region. See See *LIGGGHTS* manual. By default, the region of insertion is the entire simulation box.

– freq [‘once’]: an integer (or ‘once’) specifying how often to insert particles.

– all_in [‘yes’]: specifies whether all centers of mass of the particles inserted must be within the defined region boundaries or not. Can be either ‘yes’ or ‘no’.

– rate: a number indicating the rate of insertion, must be supplid when *mech* is ‘by_stream’ or ‘by_rate’.

  - rate_type: a string defining the type of insertion by rate, can be either ‘particlerate’ or ‘massrate’.
  - omega [(‘constant’,0,0,0)]: a tuple specifying the angular velocities: (‘constant’, omegax, omegay, omegaz).
  - orientation: an optional argument, a string that defines the orientation of non-spherical particles, can be ‘random’ or ‘template’ or ‘constant q1 q2 q3 q4’. See *LIGGGHTS* manual.
  - set_property: an optional argument, a tuple of size 2, with the 1st item being a string that defines the variable name of a fix property/atom holding a scalar value for each particle, and the 2nd item is the new value used to initialize the property upon insertion.

- createParticles(type, style, *args): Creates particles of type ’type’ (1,2, ...) using style ‘box’, ‘region’, ‘single’, or ‘random’). args can contain the additional keywords: ’basis’, ’remap’, ’units’, or ’all_in’. See *LIGGGHTS* manual.

- remove(id): remove a fix ‘id’ that is returned when fixing an insertion, an external force, a moving wall, etc.

4.2.2 Input/output methods

- dumpSetup(name=‘dump’): specifies which data to write as output. If supplied, ‘name’ is a string that specifies the dump ID which is returned by this function. By default name is ‘dump’.

- extractCoords: returns all particle positions as a numpy array.

4.2.3 Virtual and surface walls

In addition to these functions, the simulation class provides the following methods:

- setupWall: creates virtual or mesh (surface) walls
- moveMesh: specifies the motion of a surface wall
- region: creates a region

4.2.4 External body forces

- add_viscous(**args): adds a force proportional to the particle velocity. args keywords: gamma (viscosity coefficient), scale: optional arg that scales the force (default 1), species: optional argument defining the species to apply the force to (1,2,...), the default is 'all'.

4.2.5 Time integration

LIGGGHTS uses the leapfrog integration scheme to solve Newton's equations in time. The function *DEM.run(nsteps, dt, itype)* is used by *PyGran.Simulator* to create one or more integrators for all dynamical species. The three arguments for this function are:

- nsteps: number of timesteps to run (integer)
- dt: incremental timestep (float)
- itype: an optional argument that is a string or a list of strings. The string must specify the integrator type to be 'nve/sphere', 'multisphere', etc. The length of the string must be equal to all the different particle species created in the simulation, and each list item can also contains additional arguments such as: 'nve/limit relative 0.1'. See the *LIGGGHTS*

manual for all types of integrators. By default, *PyGran* assigns the right integrator type based on the particle/species types defined to create the DEM simulation class.

4.3 Examples

4.3.1 Hopper flow

LIGGGHTS supports the simulation of particles interacting with static or moving meshes. The script shown in (4.1) shows how *PyGran.Simulator* can be used to run a typial DEM simulation.

```
from PyGran import Simulator
from PyGran.Materials import glass, stearicAcid

params = {

  # Setup model + DEM engine
  'engine': Simulator.engines.liggghts,
  'model': Simulator.models.SpringDashpot,

  # Define the system
  'boundary': ('f','f','f'),
  'box':  (-1e-3, 1e-3, -1e-3, 1e-3, 0, 4e-3),

  # Define component(s)
  'SS': ({'material': stearicAcid, 'radius': ('gaussian number', 5e-5, 5
    e-6)},
    ),

  # Setup I/O params -- save particles + blade files every 1000 steps
  'traj': {'freq':1000, 'pfile': 'traj.dump', 'mfile': [None, 'blade*.
    vtk']},

  # Define computational parameters
  'nns_skin': 1e-3,
  'dt': 1e-6,
  # Apply a gravitional force in the negative direction along the z−axis
  'gravity': (9.81, 0, 0, −1),

  # Import hopper + blade meshes
  'mesh': {
    'hopper': {'file': 'silo.stl', 'mtype': 'mesh/surface', 'material':
    glass, 'args': ('scale 1e−3',)},
    'blade': {'file': 'blade.stl', 'mtype': 'mesh/surface', 'material':
    glass, 'args': ('move 0 0 1.0', 'scale 1e−3',)},
  },

  # Stage runs
  'stages': {'insertion': 1e4, 'run': 5e4},
}

# Instantiate a DEM class
sim = Simulator.DEM(**params)

# Setup a stopper wall along the xoy plane
stopper = sim.setupWalls(species=1, wtype='primitive', plane = 'zplane',
     peq = 0.0)

# Rotate blade
moveBlade = sim.moveMesh(name='blade', *('rotate origin 0. 0. 0.', 'axis
      0. 0. 1.', 'period 5e−2'))

# Insert 1000 particles in a block region
insert = sim.insert(species=1, region=('block', −5e−4, 5e−4, −5e−4, 5e
    −4, 2e−3, 3e−3), value=1000)

# Evolve the system (insertion)
sim.run(pDict['stages']['insertion'], pDict['dt'])

# Remove stopper and evolve the system
sim.remove(insert)
sim.run(pDict['stages']['run'], pDict['dt'])
```

Listing 4.1: A Python code that shows how the *LIGGGHTS* engine in *PyGran.Simulator* can be used to run a simulation.

4.3.2 Binary system

LIGGGHTS supports the simulation of multi-component systems. *PyGran.Simulator* provides a simple interface for simulating multi-component systems as shown in code (4.2).

```
from PyGran import Simulator
from PyGran.Materials import stearicAcid, glass
import numpy as np

# Create a dictionary of physical parameters
params = {

  # Define the system
  'boundary': ('p','p','f'),
  'box':  (-0.00025, 0.00025, -0.00025, 0.00025, 0, 0.002),

  # Define component(s)
  'SS': ({'material': stearicAcid, 'radius': ('gaussian number', 25.00e
     -6, 4.7e-6)},
     {'material': glass, 'radius': ('gaussian number', 25.00e-6, 4.7e-6)}
     ),

  # Setup I/O
  'traj': {'freq': 100000, 'pfile': 'traj.dump', 'mfile':'mesh-*.vtk'},
  'output': 'Binary',
  'restart': (5000, 'restart', 'restart.binary', False, None),

  # Define (optional) timestep to be used later
  'dt': 2e-9,

  # Apply gravitional force in the negative direction along the z-axis
  'gravity': (9.81, 0, 0, -1),

  # Stage runs
  'stages': {'insertion': 1e8, 'run': 1e7, 'relax': 0e5},

  # Meshes
  'mesh': {
    'wall': {'file': 'square.stl', 'mtype': 'mesh/surface/stress', '
    material': stearicAcid, 'args': ('scale 2.5e-4', 'move 0 0 -8e-4')
    },
  },
}

# Create an instance of the DEM class
sim = Simulator.DEM(**params)

# Insert all particles throughout the sim box
insert = sim.insert(species='all')

# Run the simulation
sim.run(pDict['stages']['insertion'], pDict['dt'])

# Remove insertion
sim.remove(insert)

# Setup params for vibrating mesh
freq = 40 * 2 * np.pi
nTaps, period = 100, 1.0 / freq
nSteps = period / (pDict['dt'])
relax = True

for i in range(nTaps):
  sim.moveMesh('wall', *('viblin', 'axis 0 0 1', 'order 1', 'amplitude
    12.5e-6', 'phase 0' , 'period {}'.format(period)))
  sim.run(nSteps, pDict['dt'])

  sim.remove('moveMesh')

  # Relax the system if requested by the user
  if relax:
    sim.run(nSteps, pDict['dt'])
```

Listing 4.2: A Python code that shows how the *LIGGGHTS* engine in *PyGran.Simulator* can be used to simulate a binary system.

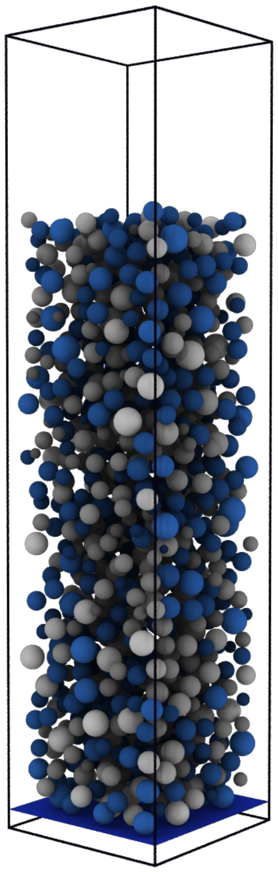


Figure 4.1: Snapshot of a two-component system: particles freely falling and colliding with a bottom wall (2D mesh) used to tap the system.

Part III

Analyzer

Chapter 5

Overview

The *Analyzer* module enables programmers to read and analyze DEM trajectory files an in intuitive way. The most fundamental class in this module is *System*, which uses a factory class to instantiate a *SubSystem* subclass. In principle, any implementation of *SubSystem* can be instantiated with this factory. A *System* class is always instantiated by passing the filename of a specific *SubSystem*:

```
Granular = System(SubSystem='path/to/file')
```

The *System* is an iterator. Thus, it can be iterated/looped over when the supplied *SubSystem* contains a time series. For example, the statement

```
for frame in System(SubSystem='path/to/file'):
```

loops over every frame stored in the suppled file, returning the frame number at each instant.

Talk a bit about dynamic attributes and else is exposed. Talk about the data dictionary. Talk about read-only data structure.

PyGran provides two *SubSystem* classes for reading particles and meshes, respectively: *System.Particles* and *System.Mesh*. These two classes are discussed in detail in the next chapters.

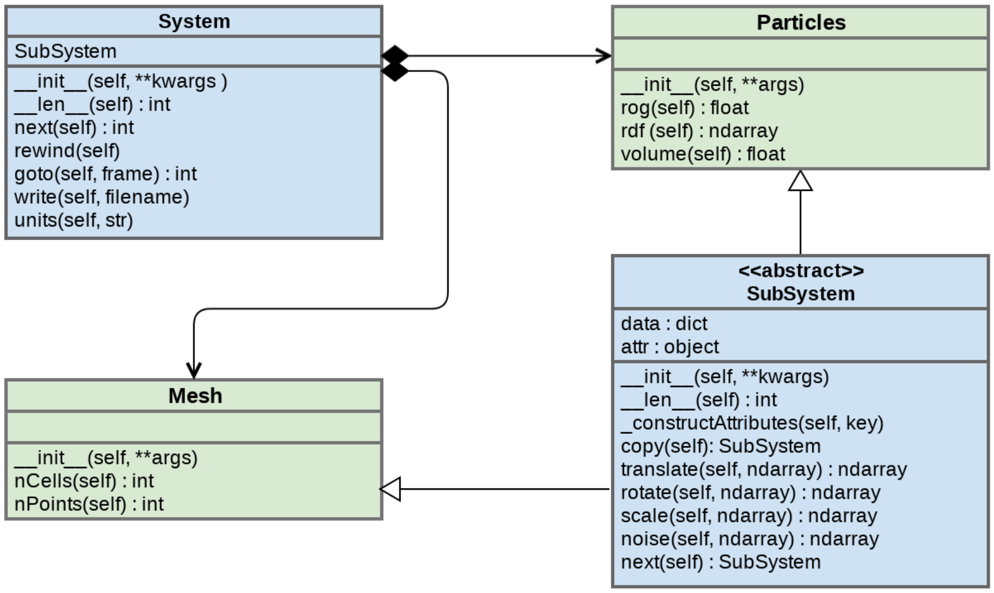


Figure 5.1: A UML diagram of the three fundamental objects and some of their methods and attributes in the *Analyzer* module. *SubSystem* contains dynamic attributes (*attr*) that are known only during runtime.

Chapter 6

Particles

The *Analyzer.Particles* class provides

6.1 Structural analysis

An example script that uses *PyGran.Analyzer* to compute the coordination number (Fig. (6.1) and the radial distribution function (Fig. (6.2)) for a granular system is shown below.

```
from PyGran import Analyzer

# Create a granular object from a LIGGGHTS dump file
Sys = Analyzer.System(Particles='traj.dump', units='micro')

# Compute the radial distribution function
g, r, _ = Sys.Particles.rdf()

# Construct a class for nearest neighbor searching
Neigh = Analyzer.equilibrium.Neighbors(Sys.Particles)

# Extract coordination number per particle
```

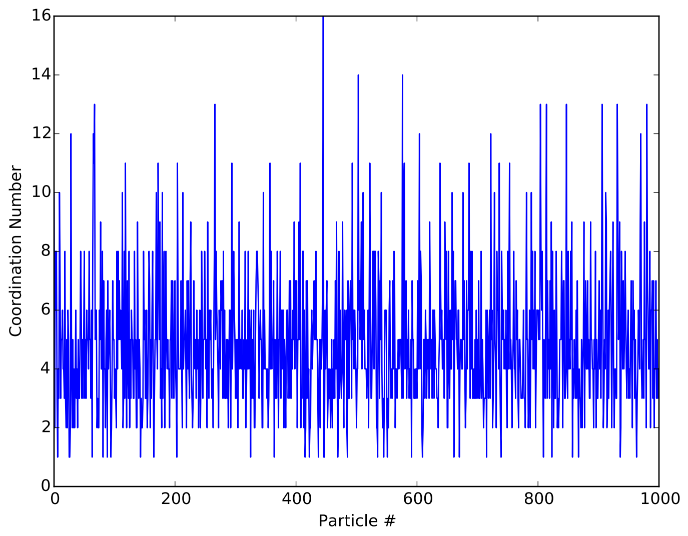


Figure 6.1: A histogram generated from code (6.1) for the coordination number of a group of particles of mean particle radius $\langle R \rangle = 50$ microns.

```
coon = Neigh.coon
```

Listing 6.1: A Python code that shows how *PyGran* can be used to do standard spatial analysis of DEM systems.

6.2 Temporal analysis

An example script that uses *PyGran.Analyzer* to compute the mass density and flow rate for a granular system is shown below.

```
from PyGran import Analyzer

# Material propreties and simulation parameters for glass
tDensity, timestep = 2500.0, 1e-6

# Create a PyGran System from a series of ESyS (trajectory) files
Gran = Analyzer.System(Particles='/home/levnon/Desktop/compaction/out-
    SpringDashpot/traj/traj.dump')
```

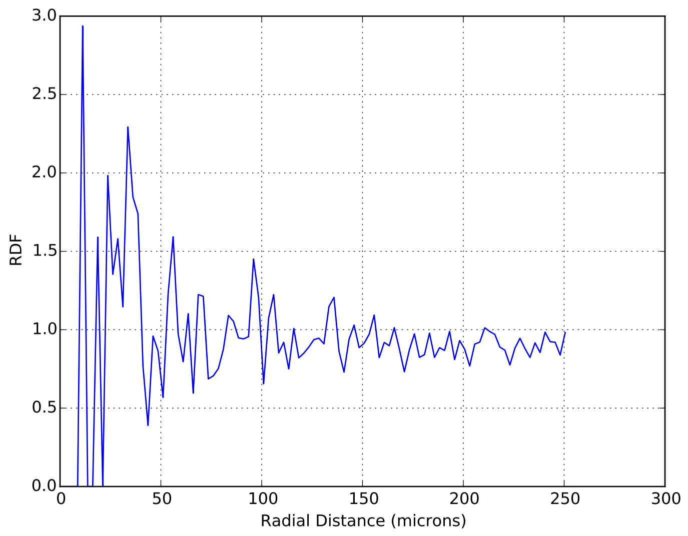


Figure 6.2: The radial distribution function (rdf) for a group of particles of mean particle radius $\langle R \rangle = 50$ microns.

```
# Each Granular object contains a Particle object that can be sliced
Parts = Gran.Particles

# Compute the mass flow rate using the 1st 10 frames
Temp = Analyzer.dynamics.Temporal(Gran)
rate = Temp.computeFlow(density=tDensity, dt=timestep)

# Compute the bulk density as a time series by looping over the Gran
    trajectory
density = []

for ts in Gran:
  density.append(Parts.density(tDensity))
```

Listing 6.2: A Python code that shows how *PyGran* can be used to do temporal analysis of DEM systems.

Chapter 7

Mesh

VTK library, support for file extensions, core attributes, bla bla bla

7.1 Particle-mesh analysis

Typically, DEM simulations consist of a set of particles interacting with 1 or more mesh(es). When coupled with *Particles*, the *Mesh* object enables a quick and easy way to analyze such simulations. An example script is shown below for a uni-axial

7.2 Coupled CFD-DEM simulations

Coupled CFD-DEM simulations are being increasingly employed in the industry to study fluidized beds [3, 4]. A sample *PyGran* script for analyzing a fluidized bed simulated with *LIGGGHTS* [5] and *OpenFOAM* [6] is shown in code 7.1. The script reads the particle (dump) trajectory file and the fluid (vtk) trajectory file to compute the pressure drop, inlet velocity, and the bed velocity along the direction of motion (z-axis).

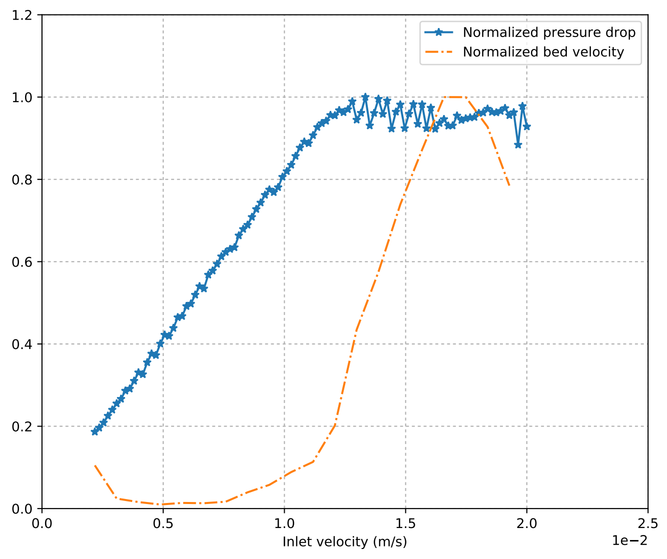


Figure 7.1: The normalized pressure drop and velocity of a fluidized bed (tutorial adopted from [7]) computed with *PyGran* as shown in code (7.1).

```
from PyGran.Analyzer import System
from scipy.linalg import norm

# Create a mesh trajectory file for the inlet & outlet files
inlet, outlet = 'CFD/inlet_*.vtk', 'CFD/outlet_*.vtk'
Traj = System(Particles='DEM/*.dump', Mesh=[inlet, outlet], vtk_type='
     poly')

# Compute mesh surface areas
iMesh, oMesh = Traj.Mesh
iArea, oArea = iMesh.CellArea.sum(), oMesh.CellArea.sum()

# Loop over inlet trajectory and compute the inlet pressure & vel
for i, timestep in enumerate(Traj):

  # Compute the weghted-average pressure inlet + outlet
  iPress[i].append((iMesh.p * iMesh.CellArea).sum() / iArea)
  oPress[i].append((oMesh.p * oMesh.CellArea).sum() / oArea)
```

```
# Compute the weighted-average inlet velocity
iVel[i].append(norm((iMesh.U.T * iMesh.CellArea).sum(axis=1) / iArea))

# Compute mean particle position along the z-azis
zMean[i].append(Traj.Particles.z.mean())
```

Listing 7.1: A Python code that shows how *PyGran* can be used to analyze coupled CFD-DEM simulations.

Chapter 8

Advanced techniques: extensions and custom objects

PyGran's extensible and object-oriented design makes it ideal for creating user-defined particles. Since *System* uses a Factory class to instantiate a *Particles* or *Mesh* object, it can in principle be used to instantiate a user-defined class. This is demonstrated in the next section for a simple coarse-grained class that demonstrates the use of the *filter* method to eliminate particles overlapping by a certain %.

8.1 coarse-graining

A simple user-defined *CoarseParticles* class can be defined as a subclass of *Particles* with two key arguments: 'scale', which controls the level of coarse-graining (or reduction) and 'percent' which is used to eliminate the resultant coarse-grained particles overlapping by a certain percentage with respect to their radius. A script that implements this class is shown below.

```
from PyGran import Analyzer

class CoarseParticles(Analyzer.Particles):
  def __init__(self, **args):
    super(CoarseParticles, self).__init__(**args)

    if 'scale' in args and 'percent' in args:
      self.scale(args['scale'], ('radius',))
      CG = Analyzer.equilibrium.Neighbors(self).filter(percent=args['
    percent'])

      self.__init__(CoarseParticles=CG)

if __name__ == '__main__':
  Traj = Analyzer.System(CoarseParticles='traj.dump', units='micro',
    scale=3, percent=25.0)
  Traj.CoarseParticles.write('CG.dump')
```

The *CoarseParticles* object uses a recursive call to instantiate a derivative of the *Particles* class and therefore inherits all of the latter's properties and methods.

Chapter 9

Plotting 2D data

In this chapter, we cover the 2D plotting functions available in *PyGran*, which uses the *matplotlib* library for producing publication quality figures. The *PyGran.Visualizer.plot2D* module provides 3 methods for plotting 2D data:

- quiver,
- pcolor(Particles, value): a function that produces a pseudocolor plot
- timePlot(System, attr, **args): a function that produces a temporal plot for a trajectory System (*PyGran.Simulator.System* object) attribute (specified by the string 'attr'). The attribute must be a variable contained in System. By default, the arithmetic mean is used to reproduce the time series for numpy array attributes. This behavior can be changed with the 'metric' argument, a string that controls which data metric to use. Any metric available in numpy can be used (mean, min, max, etc.). This function has additional optional arguments (framei, framef) that can be used to produce the plot over a specific time interval, in addition to other arguments that control certain properties of the plot (see online manual).

Bibliography

[1] Peter A Cundall and Otto DL Strack. A discrete numerical model for granular assemblies. *geotechnique*, 29(1):47–65, 1979.

[2] Colin Thornton and Zemin Ning. A theoretical model for the stick/bounce behaviour of adhesive, elastic-plastic spheres. *Powder technology*, 99(2):154–162, 1998.

[3] Takuya Tsuji, Keizo Yabumoto, and Toshitsugu Tanaka. Spontaneous structures in three-dimensional bubbling gas-fluidized bed by parallel dem–cfd coupling simulation. *Powder Technology*, 184(2):132–140, 2008.

[4] Dalibor Jajcevic, Eva Siegmann, Charles Radeke, and Johannes G Khinast. Large-scale cfd–dem simulations of fluidized granular systems. *Chemical Engineering Science*, 98:298–310, 2013.

[5] Christoph Kloss and Christoph Goniva. Liggghts–open source discrete element simulations of granular materials based on lammps. *Supplemental Proceedings: Materials Fabrication, Properties, Characterization, and Modeling, Volume 2*, pages 781–788, 2011.

[6] Hrvoje Jasak, Aleksandar Jemcov, Zeljko Tukovic, et al. Openfoam: A c++ library for complex physics simulations. In *International workshop*

on coupled methods in numerical dynamics, volume 1000, pages 1–20. IUC Dubrovnik, Croatia, 2007.

[7] Christoph Goniva, Christoph Kloss, Alice Hager, and Stefan Pirker. An open source cfd-dem perspective. In *Proceedings of OpenFOAM Workshop, Göteborg*, pages 22–24, 2010.