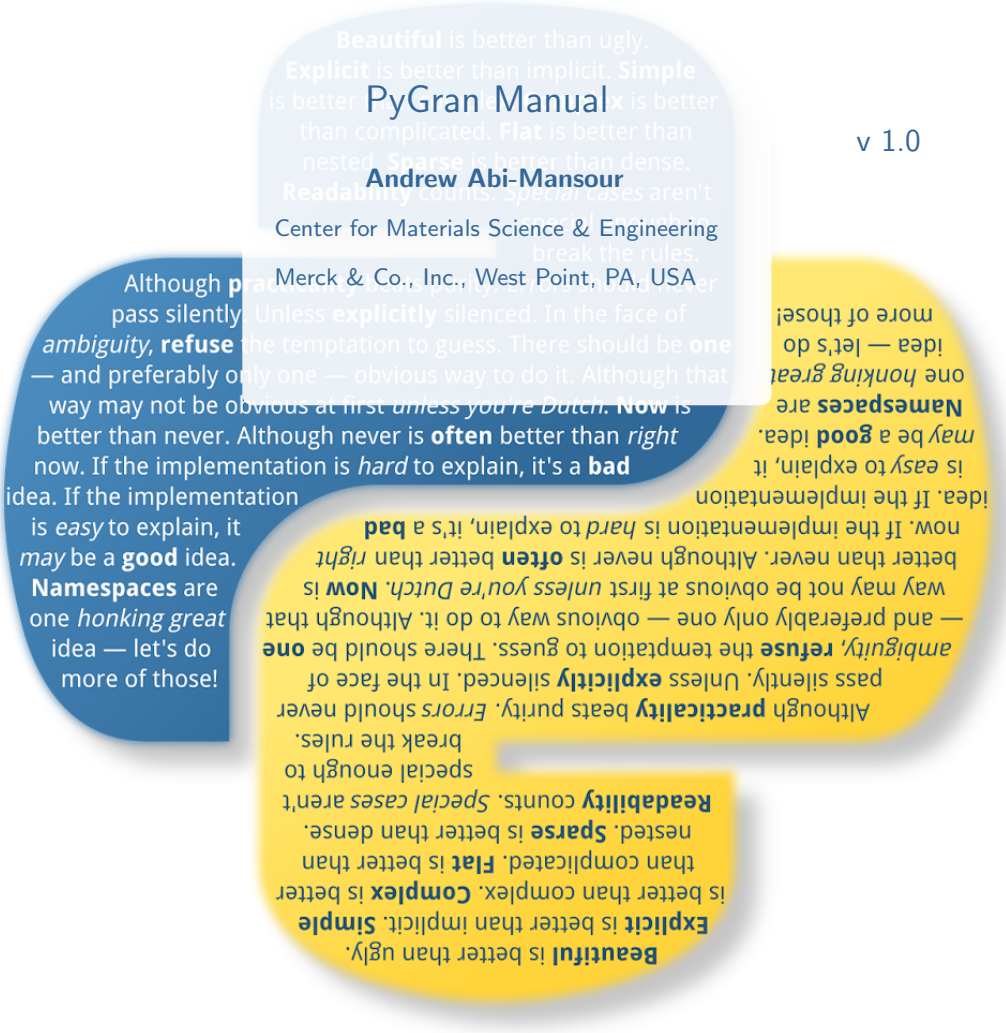
PyGran Manual

v 1.0

Andrew Abi-Mansour

Center for Materials Science & Engineering

Merck & Co., Inc., West Point, PA, USA



python™

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 of performing technical computing in DEM, enabling flexibility in how users interact with data from the onset of a simulation and until the last 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.

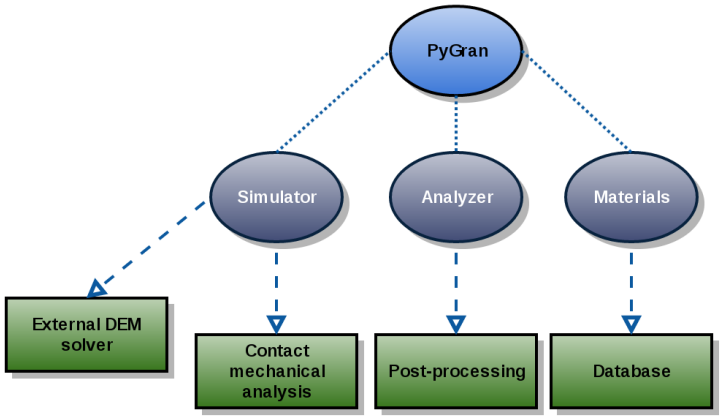


Figure 1.1: A diagram that shows the hierarchical structure of *PyGran* in terms of its core modules and submodules that can be imported from a Python input script.

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 backward 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 with mininal dependencies. The following packages must be installed before *PyGran* is configured to run:

- *Numpy* : for exposing trajectory data as *ndarray* objects and performing linear algebra floating-point operations

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.abi.mansour@merck.com

Table 2.1: Code metadata

- *Scipy* : for efficient nearest neighbor searching routines, sorting, and non-linear solvers.

2.2.2 Optional packages

For running DEM simluations with *LIGGGHTS* in parallel, OpenMPI or MPICH2 must be installed. For reading mesh files in VTK format, the VTK library must be installed. Furthermore, the following 3 Python packages must be installed for full optional functionality:

- *Cython* : for improved performance in *Analyzer.core* module
- *PyVTK* : for reading mesh files in VTK file format
- *mpi4py* : for running DEM simulations in parallel with MPI

2.3 Installation

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 'setuptools' 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.

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.

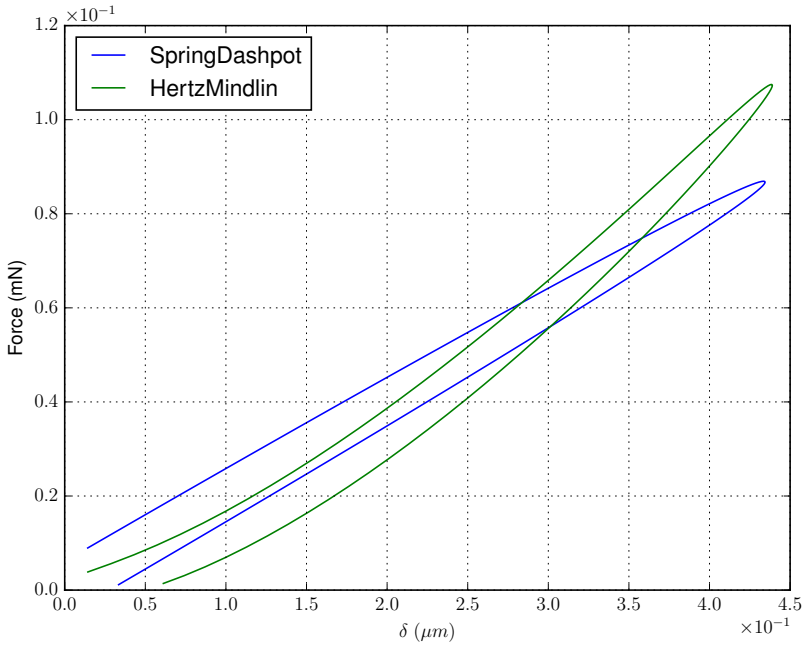


Figure 3.1: Force as a function of displacement (δ) computed for the Spring-Dashpot and Hertz-Mindlin models available in the *Simulator.models* module.

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.

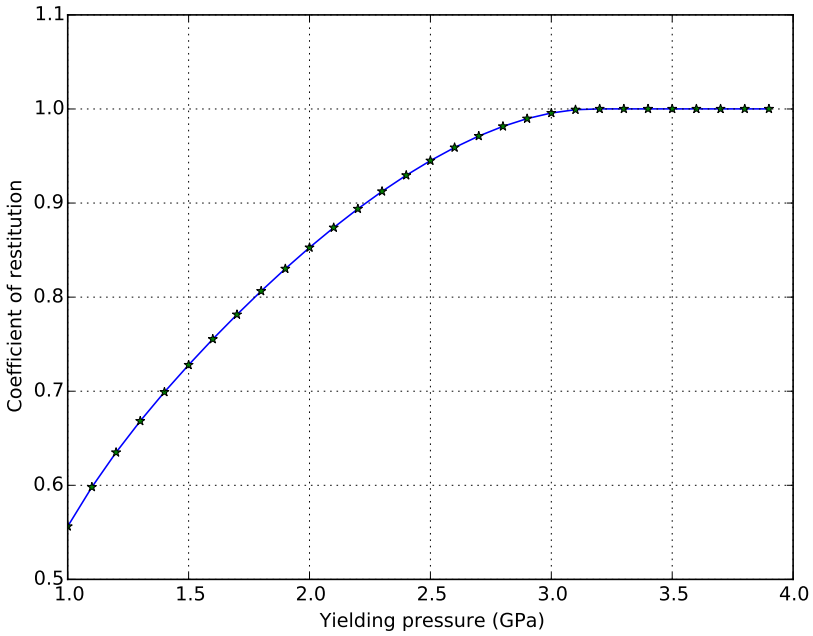


Figure 3.2: The coefficient of restitution for two spheres of reduced radius of 100 μm computed using the *ThorntonNing* model implemented in *PyGran*.

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 (shared library). An *engine* provides an interface for *PyGran* to call specific methods in the DEM solver. While *PyGran* provides an *engine* for LIGGGHTS (v 3.7), it can be readily used with solvers such as Yade and ESyS-particle that have their own Python APIs.

4.2 Example: MPI-based *LIGGGHTS* engine

4.2.1 Hopper flow

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

# Create a dictionary of physical parameters
pDict = {
    'model': Simulator.models.SpringDashpot,
```

```
    'engine': Simulator.engines.liggghts,

    # Define the system
    'boundary': ('p','p', 'f'),
    'box':  (-0.004, 0.004, -0.00012, 0.00012, -0.0001, 0.01),
    'nns_skin': 2e-4,
    'dim': 3,

    # Define particle (stearic acid) + wall (glass) components
    'SS': ({'id': 1, 'insert':'by_pack', 'natoms': 8000, \
      'material':stearicAcid, 'vol_lim': 1e-16, 'freq': 'once', \
      'radius': ('gaussian number', 70e-6, 10e-6)}, \
      {'id': 2, 'material': glass},
          ),

    # Set I/O params
    'traj': {'pfile': 'traj.dump'},

    'output': 'out-SpringDashpot',

    # Apply gravitional force in the negative direction along the z-axis
    'gravity': (98.1, 0, 0, -1),

    # Stage runs
    'dt': 2e-7,
    'stages': {'insertion': 2e5, 'run': 2e5},
    }

if __name__ == '__main__':

  # Instantiate a class based on the selected model
  pDict['model'] = pDict['model'](**pDict)

  # Create an instance of the DEM class
  sim = Simulator.DEM(**pDict['model'].params)

  # Setup a stopper wall along the xoy plane
  sim.setupWalls(name='stopper', wtype='primitive', id=2, plane = '
    zplane', peq = 0.0)

  high = 0.5e-3
```

```
scale = 30.0

for i in range(3):

  factorLow = scale * (i / (i+1.))**2.0 * high - i * high
  factorHigh = scale * ((i+1.) / (i+2.))**2.0 * high - i * high

  region = 'void{}'.format(i)
  insert = sim.insert(region, 1, *('block', -4e-3, 4e-3, -1.2e-4, 1.2e
  -4, 5e-5 + factorLow, 5e-5 + factorHigh))
  sim.run(pDict['stages']['insertion'], pDict['dt'])
  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.

4.2.2 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 ().

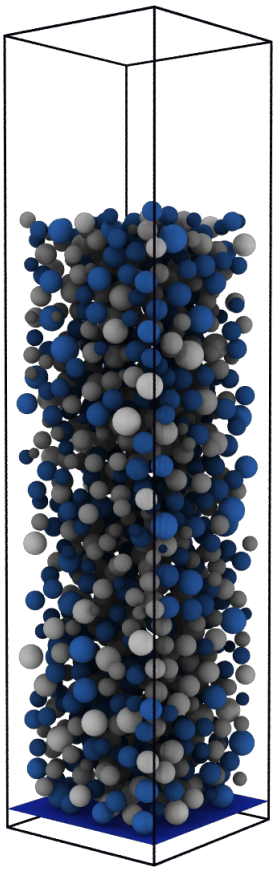


Figure 4.1: Snapshot of a two-component system.

Part III

Analyzer

Chapter 5

Particle analysis

5.1 Structural analysis

An example script that uses *PyGran.Analyzer* to compute the coordination number (Fig. (5.1) and the radial distribution function (Fig. (5.2)) 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
coon = Neigh.coon
```

Listing 5.1: A Python code that shows how *PyGran* can be used to do standard spatial analysis of DEM systems.

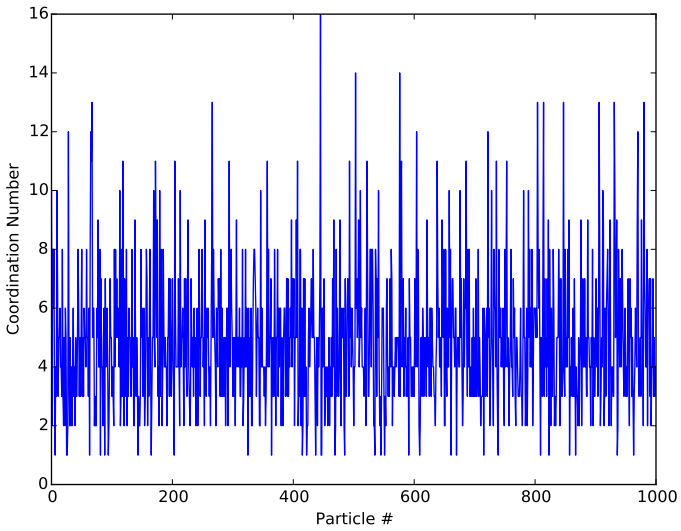


Figure 5.1: A histogram generated from code (5.1) for the coordination number of a group of particles of mean particle radius $\langle R \rangle = 50$ microns.

5.2 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')

# 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)
```

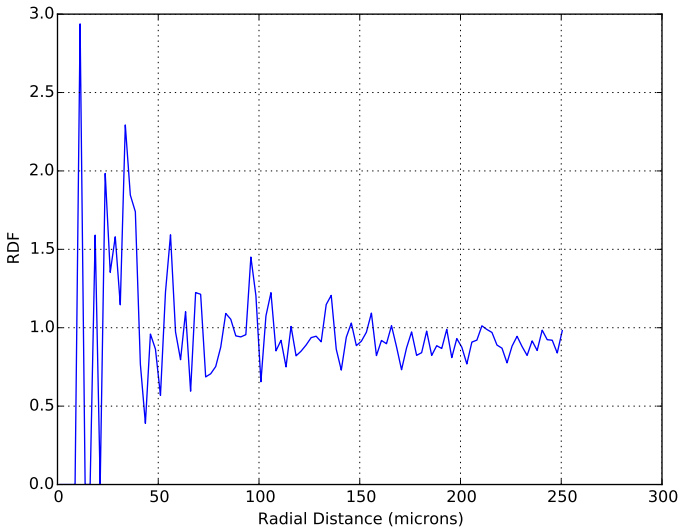


Figure 5.2: The radial distribution function (rdf) for a group of particles of mean particle radius $\langle R \rangle = 50$ microns.

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

Chapter 6

Post-processing coupled 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 6.1. 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).

```
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()
```

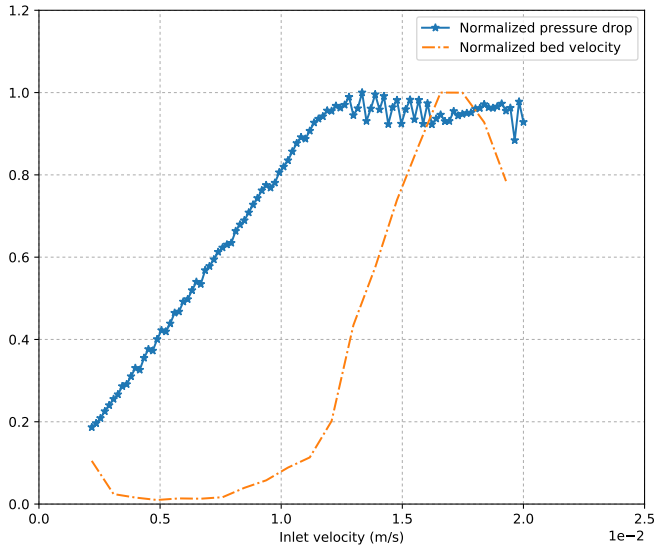


Figure 6.1: The normalized pressure drop and velocity of a fluidized bed (tutorial adopted from [7]) computed with *PyGran* as shown in code (6.1).

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

Chapter 7

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 %.

7.1 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.

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.