Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

PyGran: An object-oriented library for DEM simulation and analysis

Andrew Abi-Mansour

Center for Materials Science & Engineering, Merck & Co., Inc., West Point, PA, 19486, USA

ARTICLE INFO

Article history:
Received 8 July 2018
Received in revised form 20 January 2019
Accepted 21 January 2019

Keywords:
Granular systems
Discrete element method
Python
LIGGGHTS

ABSTRACT

PyGran is a simulation and analysis toolkit designed for particle systems with emphasis on discrete element method (DEM) simulation. *PyGran* enables DEM programmers to develop computational tools and perform interactive analysis of granular systems in Python. The toolkit provides an interface for running DEM simulations in parallel using the open-source *LIGGGHTS-PUBLIC* software, routines for performing structural and temporal analysis, and an intuitive way for building and manipulating granular systems. The object-oriented design of *PyGran* enables post-processing of coupled CFD–DEM simulation, and constructing coarse-grained representations of DEM systems. *PyGran* is released under the GNU General Public License (GPL v2.0), and its source code is available from github.

Code metadata

Current code version	1.1
Permanent link to code/repository used for this code version	http://github.com/ElsevierSoftwareX/SOFTX_2018_84
Legal Code License	GNU General Public License v2.0
Code versioning system used	Git
Software code languages, tools, and services used	python 2.7/3.5, cython 0.27, OpenMPI 1.10
Compilation requirements, operating environments & dependencies	GCC, Linux, MPI, VTK, NumPy, SciPy
Link to developer documentation/manual	Manual-v1.1
E-mail	support@pygran.com

1. Motivation and significance

The discrete or distinct element method (DEM) is a computational technique originally formulated by Cundal and Strack in 1979 for studying particle assemblies [1]. DEM is based on the application of Newtonian mechanics to a set of particles interacting via a soft-sphere pair potential [2]. DEM is becoming widely accepted today as a practical tool for addressing engineering problems in granular and discontinuous materials.

DEM has been successfully applied to problems in various areas of science and engineering such as grain processing [3,4], pharmaceutical tablet coating [5–7], rock engineering [8,9], and soil mechanics [10,11]. DEM is being increasingly employed in the industry for the design and optimization of tumbling mills [12], and it has been coupled to computational fluid dynamics [13–15] to study granular systems in which air flow is important, e.g. fluidized beds [16–18], granular flow [19,20], and die filling [21]. DEM has been expanded to include thermodynamic description and coupling to the Finite Element Method for modeling granular-continuum solid interactions [22] and characterizing the multi-scale behavior of granular media [23].

DEM simulations generate a wealth of data the interpretation of which requires analyzing time series of variables such as particle radii or positions and surface wall stresses. While there is a plethora of commercial [24,25] and open-source [26–28] DEM solvers available today, these packages offer limited post-analysis capabilities. Furthermore, implementing calibration methods [29,30] or postprocessing algorithms can be limited or impossible within existing packages [26–28,31] since that often requires an intimate knowledge of the internals of the software. *PyGran* circumvents this problem by providing a unified extensible interface that encapsulates basic classes and methods that can be used or modified in simple Python scripts for performing complex analysis.

PyGran utilizes libraries such as *SciPy* [32] and *NumPy* [33,34] that provide numerical linear algebra and optimization routines, and it provides classes that encapsulate DEM output data as *NumPy* arrays. This provides *PyGran* with a particle selection language similar to that used in VMD [35], CHARMM [36], and *MDAnalysis* [37] for macromolecular systems. For qualtitative analysis, *PyGran* provides routines for plotting DEM data using matplotlib [38].

E-mail address: anabiman@indiana.edu.

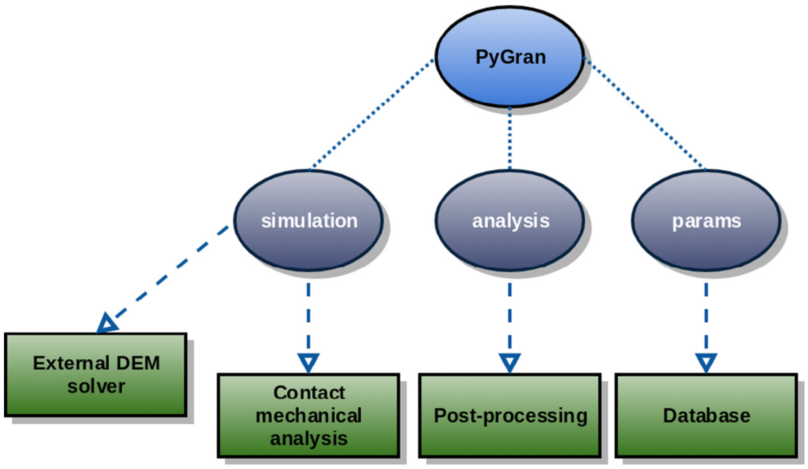

Fig. 1. A diagram that shows the hierarchical structure of *PyGran* in terms of its core modules and their associated scopes.

This paper outlines the extensible framework and core modules of *PyGran* (Section 2). Few illustrative examples on how *PyGran* can be used for analyzing DEM data are provided in Section 3. Finally, Section 4 discusses how *PyGran* can make DEM more accessible to researchers and engineers who may not necessarily have an extensive background in technical computing or programming.

2. Software description

One of the key goals in the design of *PyGran* is to enable programmers to rapidly implement their own methods for analyzing DEM systems. For that purpose, *PyGran* exposes all of its core classes and methods through a Python application programming interface (API). Python supports a variety of interpreters and development tools that make it ideal for interactive computing. *PyGran* uses Python's object-oriented features to provide an intuitive interface for building and manipulating particle systems. Furthermore, *PyGran*'s API can be readily used in conjunction with DEM solvers such as *ESyS-Particle* [27] and *Yade* [26], and it provides a module for running DEM simulation in parallel (with MPI) with *LIGGGHTS-PUBLIC* [28].

PyGran's API consists of static and dynamic interfaces. The latter enables *PyGran* to expose object attributes during runtime since these are usually not known *a priori* and in fact may vary during the course of a simulation. This dynamic interface also provides *PyGran* with support for reading custom simulation data as well as a variety of standard file formats (covered in detail in the manual).

2.1. Software architecture and functionality

The overall structure of *PyGran* is based on an object-oriented modular approach. *PyGran* consists of 3 core modules: *simulation*, *analysis*, and *params*. These modules use Python libraries such a *NumPy*, *SciPy*, and *matplotlib* to perform numerical computations and visualization of DEM data. Each of these modules can be used separately or concurrently for simulation and/or post-analysis purposes (Fig. 1). The *simulation* module provides an interactive interface for running DEM simulation with an engine (a *LIGGGHTS* engine is available in *PyGran*). This module also implements common contact mechanical models that can be used for numerical analysis. The most powerful feature of this engine is its 'single script, multiple parameter' capability, which enables users to run multiple simulations simultaneously, each with different material parameters using the same python script. This feature is efficient to use when running *LIGGGHTS* in parallel (with MPI). The *analysis* module provides submodules and functions for building and analyzing granular systems, as well as performing structural, temporal, and image analysis. The *params* module serves as an extensible database of user-defined granular materials. This database encapsulates the properties of each material in a Python dictionary which can be modified and/or imported into a *PyGran* simulation or analysis script. The specific keys and values of this Python dictionary are covered in the manual.

While the *simulation* module is discussed in detail in the manual, the focus of this section and remaining parts of the paper is the *analysis* module which provides core classes central to post-processing and analyzing DEM simulation data (Fig. 2).

2.1.1. System

System stores all the objects, methods, and properties that describe the state of a DEM system. It is therefore the most fundamental class in the *analysis* module, and its constructor uses a factory to create instances of classes available in *PyGran* or defined by the user. Usually these classes are derived from *SubSystem* to represent basic elements in a DEM simulation, such as particles and surface triangulations (representing walls). *PyGran* therefore provides the *Particles* and *Mesh* classes (subclasses of *SubSystem*) to represent these elements, which are stored in *System*. The general expression for constructing a *System* object is:

```
System_obj = System(Element=val, ..., Elements=[vals])
```

Here the keyword arguments *Element*, …, *Elements* are classes that are either implemented in *PyGran* (such as *Particles* or *Mesh*), or defined by the user in a separate module or script. This generic way of instantiating *System* enables users to easily create their own classes that represent different aspects of a DEM simulation such as superquadric [39] or coarse-grained [40] particles. The input *val* is either a *string* that provides the path to the input trajectory file or an object of type *Element*. Likewise, [*vals*] is a list of either *strings* (when reading a list of trajectory files) or objects of type *Elements*. For example, a *System* object can be created with the following statement:

```
System_obj = System(Particles=path_to_pfile, Mesh=path_to_mfile
    )
```

In this example, the *System* constructor creates a new *Particles* object that stores all particle properties read from the input trajectory file, and a new *Mesh* object that stores mesh nodes and other properties read from the input mesh file. The particles and mesh(es) are accessed via *System_obj.Particles* and *System_obj.Mesh*, respectively.

System is an *iterator* that handles I/O operations and ensures proper frame to frame propagation when reading input trajectory

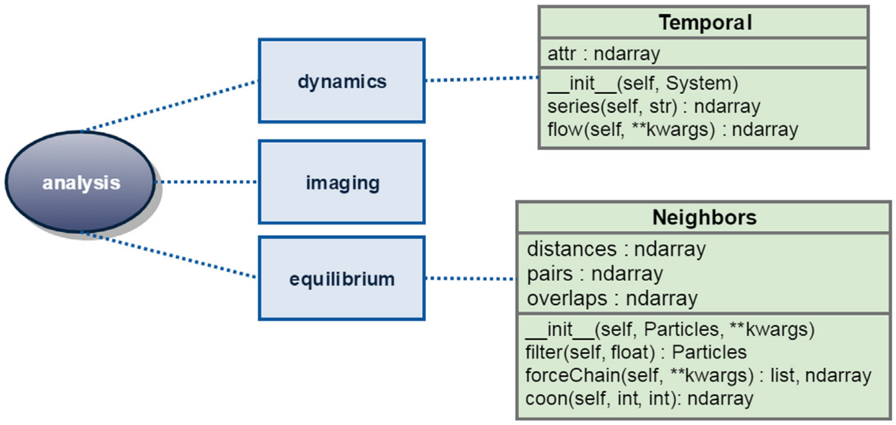
analysis
dynamics
imaging
equilibrium
Temporal
attr : ndarray
__init__(self, System)
series(self, str) : ndarray
flow(self, **kwargs) : ndarray
Neighbors
distances : ndarray
pairs : ndarray
overlaps : ndarray
__init__(self, Particles, **kwargs)
filter(self, float) : Particles
forceChain(self, **kwargs) : list, ndarray
coon(self, int, int): ndarray

Fig. 2. The *analysis* module in *PyGran* contains 3 submodules: *dynamics* which provides the *Temporal* class for time series analysis, *imaging* which provides methods for image analysis, and *equilibrium* which provides the *Neighbors* class for structural analysis.

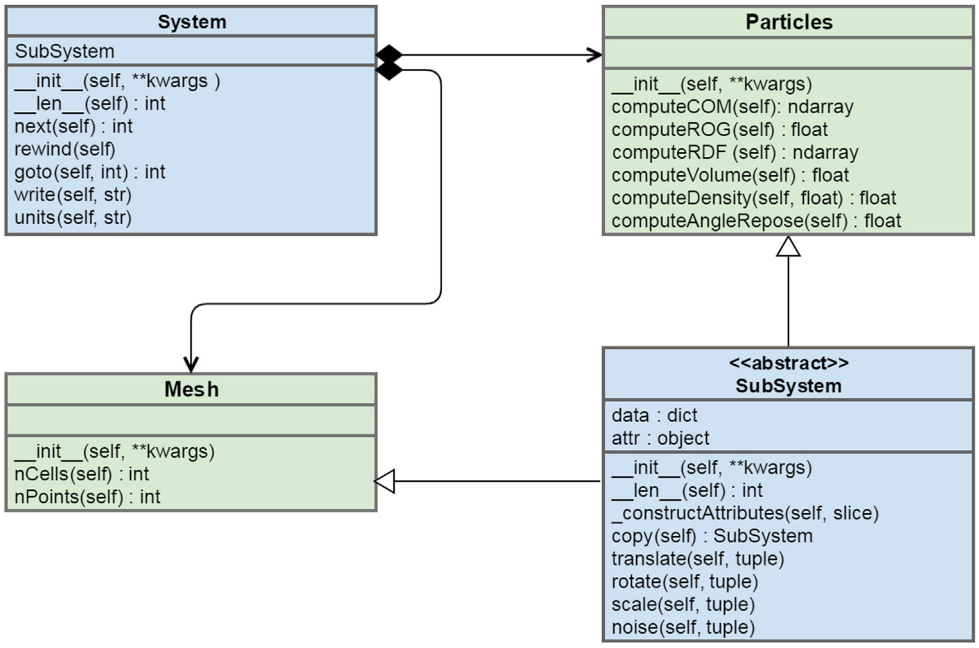

Fig. 3. A UML diagram of the fundamental classes and some of their methods and attributes in the *analysis* module. The state of a DEM system is determined by the *System* class, which creates one or more instances of classes derived from *SubSystem* that describe basic DEM elements such as particles or surface mesh(es).

Table 1
The 4 different unit systems used in *PyGran* for reading and writing DEM data.

System / Variable	Length	Time	Mass
S.I.	m	s	Kg
Micro	µm	µs	µg
CGS	cm	s	g
Nano	nm	ns	ng

files. For efficiency, only a single frame is loaded in memory at any given time. This class implements a *__next__* function that advances the state of the system to the next frame and returns the current loaded frame number. Additional functions such as *goto* and *rewind* are also provided by *System* for frame propagation. Fig. 3 summarizes the UML diagrams of the core classes in the *analysis* module, and it lists some of the static attributes each class contains, as well as the dynamic attributes (*attr*) that are known only during runtime. The 4 different unit systems supported by this class are summarized in Table 1.

2.1.2. Subsystem

This is an abstract class that encapsulates common attributes and methods for basic DEM 'elements' such as *Particles* and *Mesh*, both being derived classes of *SubSystem*. In this context, an 'element' represents a basic DEM unit such as a particle or a (surface mesh) node. The length of a *SubSystem* object is the total number of elements it stores. While *SubSystem* is mutable, its attributes cannot be directly modified by the user, i.e. they can modified only by the methods in *SubSystem*.

Instantiation and slicing

SubSystem objects can be instantiated using a Python dictionary that contains all the attributes (such as nodes, positions, velocities, ...) that define DEM 'elements'. These are usually read from an input trajectory file, or supplied by the user for building particle systems. The basic data structure in this class is a Python dictionary (*SubSystem.data*) which stores DEM-related attributes and is used to generate the interface of a *SubSystem* object during runtime. For example, the expression *SubSystem(data={'x':ndarray})* creates a

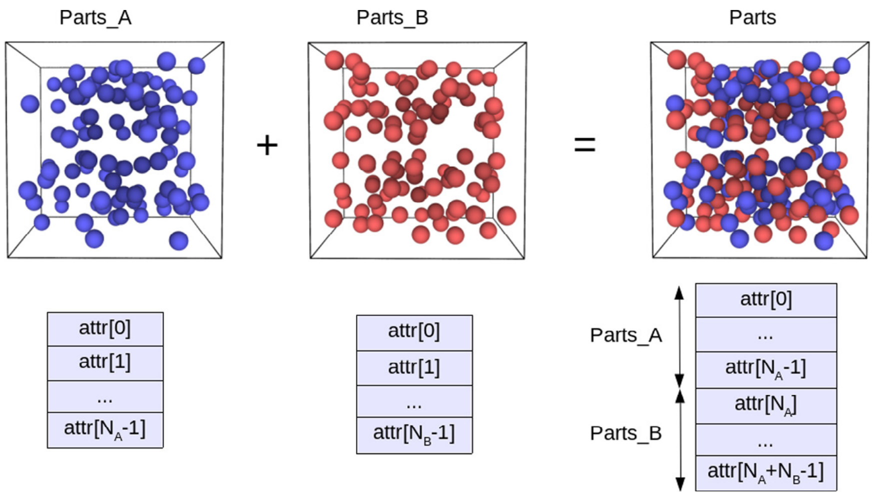

Fig. 4. The addition of two *Particles* objects (Parts_A and Parts_B) with the '+' operator results in a new *Particles* object (Parts) that concatenates and stores array attributes both objects have in common.

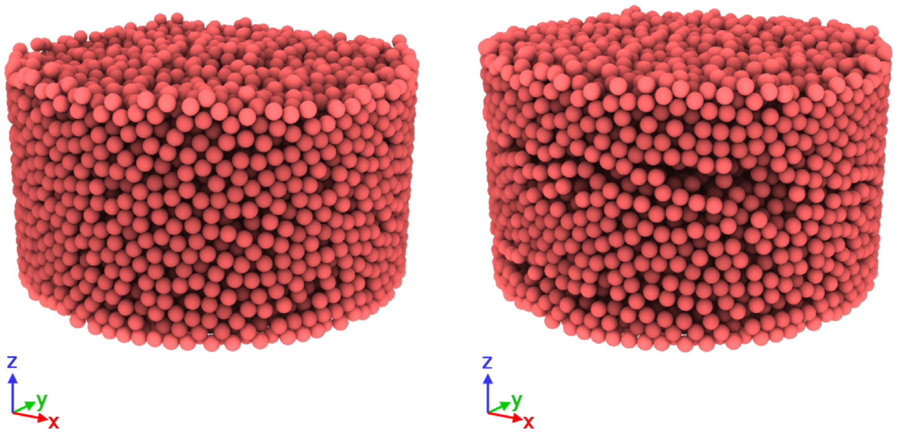

Fig. 5. Snapshots of a cylindrical bed consisting of 10,000 spherical particles characterized by a minimum fluidization velocity $v_f \approx 10$ mm/s and subjected to a fluid flow of velocity $v \in [0, 20]$ mm/s. The snapshot on the left shows the particles in static state where $v < v_f$ ($v \in [0, 10)$ mm/s), while the snapshot on the right shows the particles in fluidized state where $v > v_f$ ($v = 20$ mm/s).

new *SubSystem* object (*SS_obj*) that stores 'x' *ndarray* (*NumPy* array) accessed via *SS_obj.x*.

SubSystem objects can be created and manipulated in ways similar to those of *NumPy* arrays; the general syntax for slicing a *SubSystem* object is

```
SliceSub = SS_obj[sel]
```

where *sel* is an integer, a *NumPy* array (of type *int* or *bool*), or a Python *slice*. For example, if *sel* is an integer *i*, then *SliceSub* becomes a single *SubSystem* object (or an element) containing attributes of the *i*th entry in *SS_obj*. Similarly, if *sel* is *slice(i,j)* (i.e. $sel = i : j$), then the resultant *SliceSub* becomes a *SubSystem* object containing entries *i* to *j* in *SS_obj*. If *Parts* is a *SubSystem* object of the *Particles* type, and *sel* is the boolean array $Parts.radius > value$, then the expression *Parts[sel]* returns a new *Particles* object which contains every particle whose radius is greater than *value*. More complex selections that involve multiple conditional statements can be created with the bitwise '|' and '&' operators. For instance, a boolean array that selects all non-static particles along the z direction or those in a cylindrical region ($z > 0$) of maximum height h, radius r, and center $(0, 0, h/2)$ can be constructed as follows

```
sel = (Parts.vz != 0) | ((Parts.z <= h) & (Parts.x**2 + Parts.y
    **2 <= r**2))
```

A sliced class can then be instantiated as before with the *Parts[sel]* expression.

Generators

SubSystem objects are iterable. Thus, looping over a *SubSystem* object is equivalent to looping over all of its stored elements such that the attributes of each can be accessed but not modified, i.e.

```
for element in SS_obj:
    # Access element.attribute
```

Here 'element' is a *SubSystem* object that stores all of the attributes in *SS_obj* corresponding to the target element. For instance, if *SS_obj* contains a *NumPy* array called 'attribute', then for an integer *i* less than the length of this array, *SS_obj[i].attribute* is equivalent to *SS_obj.attribute[i]*.

2.1.3. Particles

A granular system consists of a set of particles represented by *Particles*, a subclass of *SubSystem*, which can be delineated using *PyGran*'s granular selection language (Section 2.1.2).

Binary operations

Extended assignments can be made to *Particles* objects with the '+=' operator. The array attributes stored in the left operand are appended if they are found in the right operand. The length of the left operand increases by the number of elements stored in the right operand. For example, if *Parts_A* and *Parts_B* are 2 *Particles* objects containing N_A and N_B particles, respectively, then the former can be appended to the latter with the following statement:

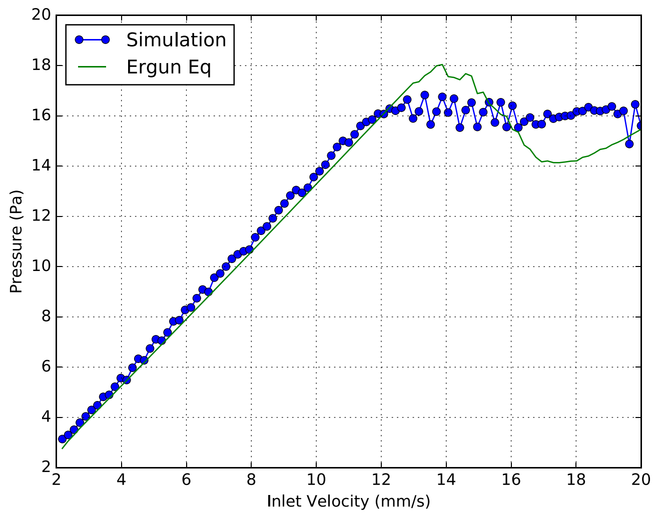

Fig. 6. The pressure drop as a function of the inlet velocity in a fluidized bed computed with *PyGran* as shown in code 1. The numerical solution for the pressure drop is compared to the analytical solution obtained from the Ergun equation [41].

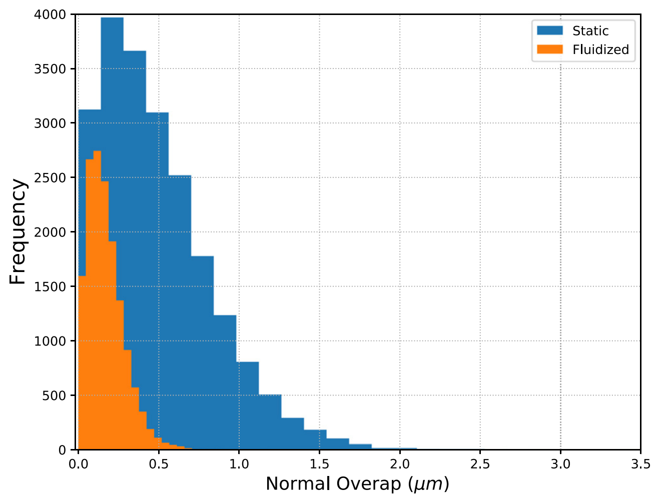

Fig. 7. A histogram of the particle–particle normal overlaps from code 1 for a granular system in static and fluidized states.

```
Parts_B += Parts_A
```

If *Parts_A* has fewer attributes than those in *Parts_B*, then this assignment is rejected. Otherwise, any additional attributes in *Parts_A* not found in *Parts_B* are neglected. After assignment, *len(Parts_B)* $= N_A + N_B$.

Parts_A and *Parts_B* can also be concatenated with the '+' operator. This operation can lead to reduction in the number of resultant attributes if one of the objects being added has attributes not found in the other. Therefore, the resultant object always acquires concatenated attributes that are common to both objects being added. For example, a new *Particles* object (*Parts*) containing $N_A + N_B$ particles can be created with the following statement:

```
Parts = Parts_A + Parts_B
```

The array attributes found in both objects are sequentially concatenated and stored in *Parts* as shown in Fig. 4, and *len(Parts)* $= N_A + N_B$.

Basic methods

Some of the basic methods in *Particles* are shown in Fig. 3. Furthermore, the *analysis* module provides a *Neighbors* class for performing efficient nearest neighbor analysis. With this class, properties such as coordination numbers, displacements, and force chains can be readily computed (Section 3.1).

Input/output

Any class derived from *SubSystem* must implement read/write methods. In the current version, *PyGran* supports reading and writing particle trajectory files for *LIGGGHTS*. The input trajectory can be a dump or a vtk file.

2.1.4. Mesh

Surface walls are represented in *PyGran* by the *System.Mesh* class, a subclass of *Subsystem* (Fig. 3). This class uses the VTK library [42] to read a mesh trajectory file and exposes all of its stored attributes to the user. This is particularly useful for analyzing coupled CFD–DEM simulations as shown in Section 3.1.

3. Illustrative examples

3.1. Fluidized beds and contact analysis

Coupled CFD–DEM simulations are being increasingly employed in the industry to study fluidized beds [16,17] as well as the effect of air on powder flow [21,43]. A sample *PyGran* script for analyzing a fluidized bed (Fig. 5) simulated with *LIGGGHTS-PUBLIC* [28] and *OpenFOAM* [44] is shown in code 1. A total of 10,000 spherical particles of diameter 1 mm are inserted under gravity in the negative *z* direction, in a cylinder of diameter 27.6 mm and height 553 mm. An inlet of uniform (fixed) velocity (0 − 20 mm/s) is imposed at the bottom of the cylinder. The flow is assumed to be incompressible along the direction of motion (*z*-axis), and the fluid mass density is set to 10 g/L. A fixed pressure outlet (10 Pa) is imposed at the top of the cylinder while slip boundary conditions are applied for the fluid at the side walls. For further information on the simulation setup, see [14]. The script shown in code 1 reads the particle (dump) trajectory file and the fluid (vtk) trajectory files to compute the pressure drop across the bed as a function of the inlet velocity (Fig. 6), as well as the normal overlaps of the particles in static and fluidized states (Fig. 7).

```
from PyGran import analysis

# Setup input filenames and initial variables
vtk_files, dump_file = ['inlet_*.vtk', 'outlet_*.vtk'], 'granules.
    dump'
inlet, outlet, press_drop = 0, 1, []

# Create a system class for the bed
Bed = analysis.System(Particles=dump_file, Mesh=vtk_files)

# Construct a nearest-neighbors class
NNS = analysis.Neighbors(Bed.Particles)

# Extract normal overlaps for the static particles
overlaps_static = NNS.overlaps

# Loop over system trajectory and compute the pressure drop
for timestep in Bed:
  p_inner, p_outer = Bed.Mesh[inlet].p, Bed.Mesh[outlet].p
  press_drop.append(p_inner - p_outer)

# Extract normal overlaps for the fluidized particles
overlaps_fluid = NNS.overlaps
```

Listing 1: A Python code that shows how the *System* class in *PyGran* enables post-processing of coupled CFD–DEM simulation: the input particle (dump) and fluid mesh (vtk) trajectory files are used to compute the particle–particle overlaps and the pressure drop across the bed.

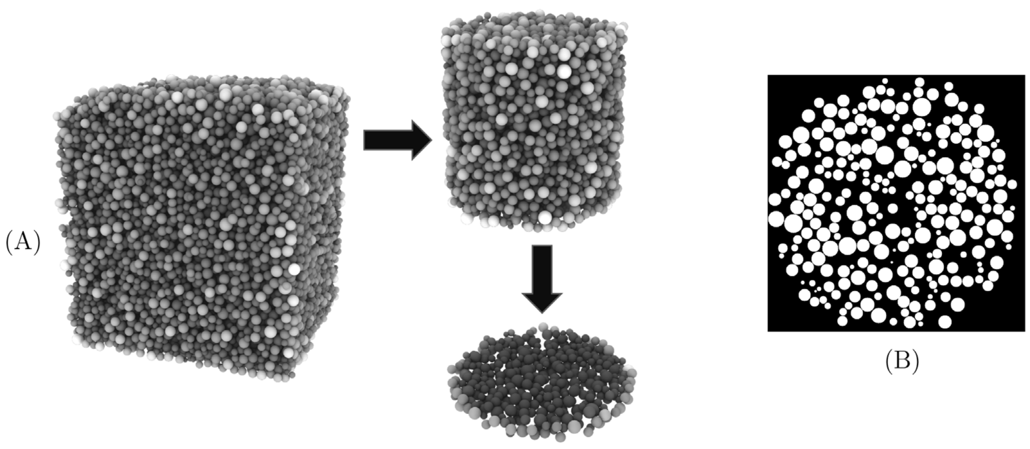

Fig. 8. The *imaging* module provides methods for generating sequence of 2D images (B) from a 3D granular configuration (A). In the example shown in (A), a cubic sample is partitioned into a cylindrical region, from which a slice of thickness 1 μm is created along the axis of the cylinder and then mapped to a 2D image (B) of resolution 1 μm/pixel as shown in code 2.

3.2. Image analysis

It is common to characterize powders with X-ray computed tomography (XRCT) [45], an experimental technique that provides information on the 3D structure of a solid sample in the form of stacked 2D images (or slices). *PyGran* provides routines for image analysis and manipulation such as slicing a DEM particle configuration into a sequence of images (Fig. 8) that can be used for comparison to XRCT data as shown in code 2.

```
from PyGran.analysis import System, imaging

# Create granular system from a dump file in S.I. units
Gran = System(Particles='traj.dump', units='si')

# Go to frame 10
Gran.goto(frame=10)

# Select a cylindrical region of radius 0.5 mm and height 1 mm
x,y,z = Gran.Particles.x, Gran.Particles.y, Gran.Particles.z
sel = (x**2 + y**2 <= 2.5e-7) & (z >= 0) & (z <= 1e-3)
Parts = Gran.Particles[sel]

# Set image resolution and create a string for saving image output
res, img = 1e-6, 'XRCT-{}.bmp'

# Slice Parts along the z-axis into images of resolution 1 micron/
    pixel
for z in arange(0, Parts.z.max() + res, res):
  imaging.slice(Parts, z, z+res, axis='z', resol=res, output=img.
      format(z))
```

Listing 2: A Python code that demonstrates how complex selections of *Particles* objects in *PyGran* can be created and used for image analysis.

4. Impact and conclusions

PyGran is a Python toolkit designed for aiding DEM researchers and practitioners in performing interactive quantitative analysis of granular and powder systems. The *simulation* module in *PyGran* provides an intuitive interface for running complex DEM simulations with *LIGGGHTS-PUBLIC* [28] in a way that is independent of the underlying DEM software. Thus, the abstraction from simulation code that *PyGran* provides makes it possible for experimentalists to run DEM simulations without having extensive background in scientific computing. This module also enables researchers to implement and investigate new contact models for simulating 2-particle collisions, and perform optimization and parametrization DEM simulations with *LIGGGHTS-PUBLIC* on clusters and supercomputers.

As shown in Section 3, the *analysis* module in *PyGran* enables application-oriented engineers to perform interactive and scripted analysis of DEM simulation data using routines geared towards particle and/or mesh manipulation and analysis. This module also provides methods for temporal analysis and structural characterization of powders, and a particle selection language for quick and efficient slicing of granular assemblies.

Since *PyGran* is an open-source toolkit, it enables developers with intimate knowledge of DEM software to implement their own methods and algorithms for DEM and coupled CFD–DEM simulation. For instance, coarse-grained and scaled-up representations of granular systems are straight forward to construct with Python's multiple inheritance feature and the *Particles* class provided by the *analysis* module. Given the extensible and object-oriented design of *PyGran* and its generic data structures, future versions of the toolkit could also support additional open-source DEM packages and more advanced features that help engineers and researchers solve complex problems in particle technology.

Acknowledgment

The author would like to acknowledge Matthew Metzger for his help in improving the manuscript.

References

[1] Cundall PA, Strack OD. A discrete numerical model for granular assemblies. Geotechnique 1979;29(1):47–65.
[2] Kruggel-Emden H, Simsek E, Rickelt S, Wirtz S, Scherer V. Review and extension of normal force models for the discrete element method. Powder Technol 2007;171(3):157–73.
[3] Tijskens E, Ramon H, De Baerdemaeker J. Discrete element modelling for process simulation in agriculture. J Sound Vib 2003;266(3):493–514.
[4] Ketterhagen WR, am Ende MT, Hancock BC. Process modeling in the pharmaceutical industry using the discrete element method. J Pharm Sci 2009;98(2):442–70.
[5] Pandey P, Song Y, Kayihan F, Turton R. Simulation of particle movement in a pan coating device using discrete element modeling and its comparison with video-imaging experiments. Powder Technol 2006;161(2):79–88.
[6] Boehling P, Toschkoff G, Just S, Knop K, Kleinebudde P, Funke A, Rehbaum H, Rajniak P, Khinast J. Simulation of a tablet coating process at different scales using DEM. Eur J Pharm Sci 2016;93:74–83.
[7] Boehling P, Toschkoff G, Knop K, Kleinebudde P, Just S, Funke A, Rehbaum H, Khinast J. Analysis of large-scale tablet coating: modeling, simulation and experiments. Eur J Pharm Sci 2016;90:14–24.
[8] Jing L. A review of techniques, advances and outstanding issues in numerical modelling for rock mechanics and rock engineering. Int J Rock Mech Min Sci 2003;40(3):283–353.
[9] Yan Y, Zhao J-F, Ji S-Y. Discrete element analysis of the influence of rock content and rock spatial distribution on shear strength of rock-soil mixtures. Gongcheng Lixue/Eng Mech 2017;34(6):146–56.

[10] Ting JM, Corkum BT, Kauffman CR, Greco C. Discrete numerical model for soil mechanics. J Geotech Eng 1989;115(3):379–98.
[11] Shmulevich I, Asaf Z, Rubinstein D. Interaction between soil and a wide cutting blade using the discrete element method. Soil Tillage Res 2007;97(1):37–50.
[12] Mishra B. A review of computer simulation of tumbling mills by the discrete element method: Part II—practical applications. Int J Miner Process 2003;71(1):95–112.
[13] Kloss C, Goniva C, Hager A, Amberger S, Pirker S. Models, algorithms and validation for opensource DEM and CFD-DEM. Prog Comput Fluid Dyn 2012;12(2–3):140–52.
[14] Goniva C, Kloss C, Hager A, Pirker S. An open source CFD-DEM perspective. In: Proceedings of OpenFOAM Workshop, Göteborg; 2010, p. 22–4.
[15] Liu D, Bu C, Chen X. Development and test of CFD-DEM model for complex geometry: a coupling algorithm for fluent and DEM. Comput Chem Eng 2013;58:260–8.
[16] Tsuji T, Yabumoto K, Tanaka T. Spontaneous structures in three-dimensional bubbling gas-fluidized bed by parallel DEM-CFD coupling simulation. Powder Technol 2008;184(2):132–40.
[17] Jajcevic D, Siegmann E, Radeke C, Khinast JG. Large-scale CFD-DEM simulations of fluidized granular systems. Chem Eng Sci 2013;98:298–310.
[18] Thornton AR, Krijgsman D, Fransen R, Briones SG, Tunuguntla DR, te Voortwis A, Luding S, Bokhove O, Weinhart T. Mercury-DPM: fast particle simulations in complex geometries. Newsl EnginSoft 2013;10(1):48–53.
[19] Neuwirth J, Antonyuk S, Heinrich S, Jacob M. CFD-DEM study and direct measurement of the granular flow in a rotor granulator. Chem Eng Sci 2013;86:151–63.
[20] Shan T, Zhao J. A coupled CFD-DEM analysis of granular flow impacting on a water reservoir. Acta Mech 2014;225(8):2449.
[21] Guo Y, Kafui K, Wu C-Y, Thornton C, Seville JP. A coupled DEM/CFD analysis of the effect of air on powder flow during die filling. AIChE J 2009;55(1):49–62.
[22] Michael M, Vogel F, Peters B. DEM-FEM coupling simulations of the interactions between a tire tread and granular terrain. Comput Methods Appl Mech Engrg 2015;289:227–48.
[23] Guo N, Zhao J. A coupled FEM/DEM approach for hierarchical multiscale modelling of granular media. Internat J Numer Methods Engrg 2014;99(11):789–818.
[24] DEM Solutions, EDEM User Guide (version 2018).
[25] CD-Adapco, STAR-CCM+ User Guide (version 13).
[26] Kozicki J, Donzé F. YADE-OPEN DEM: an open-source software using a discrete element method to simulate granular material. Eng Comput 2009;26(7):786–805.
[27] Weatherley DK, Boros VE, Hancock WR, Abe S. Scaling benchmark of ESyS-particle for elastic wave propagation simulations. In: e-Science (e-Science), 2010 IEEE sixth international conference on. IEEE; 2010, p. 277–83.
[28] Kloss C, Goniva C. LIGGGHTS-open source discrete element simulations of granular materials based on Lammps. In: Supplemental proceedings: Materials fabrication, properties, characterization, and modeling, Volume 2. 2011, p. 781–8.
[29] Rackl M, Hanley KJ. A methodical calibration procedure for discrete element models. Powder Technol 2017;307:73–83.
[30] Cheng H, Shuku T, Thoeni K, Yamamoto H. Probabilistic calibration of discrete element simulations using the sequential quasi-monte carlo filter. Granul Matter 2018;20(1):11.
[31] Stukowski A. Visualization and analysis of atomistic simulation data with OVITO-the open visualization tool. Modelling Simulation Mater Sci Eng 2009;18(1):015012.
[32] Jones E, Oliphant T, Peterson P, et al. SciPy: Open source scientific tools for Python; 2001. URL http://www.scipy.org. Online [Accessed 25 October 2018].
[33] Oliphant TE. A guide to NumPy, Vol. 1. Trelgol Publishing USA; 2006.
[34] Walt Svd, Colbert SC, Varoquaux G. The NumPy array: a structure for efficient numerical computation. Comput Sci Eng 2011;13(2):22–30.
[35] Humphrey W, Dalkea A, Schulten K. VMD: visual molecular dynamics. J Mol Graph Model 1996;14:33–8.
[36] Brooks BR, Bruccoleri RE, Olafson BD, States DJ, Swaminathan S, Karplus M. CHARMM: a program for macromolecular energy, mnimization, and dynamics calculation. J Comput Chem 1983;4:187–217.
[37] Michaud-Agrawal N, Denning EJ, Woolf TB, Beckstein O. MDAnalysis: a toolkit for the analysis of molecular dynamics simulations. J Comput Chem 2011;32(10):2319–27.
[38] Hunter JD. Matplotlib: a 2D graphics environment. Comput Sci Eng 2007;9(3):90–5.
[39] Cleary PW, Sawley ML. DEM modelling of industrial granular flows: 3D case studies and the effect of particle shape on hopper discharge. Appl Math Model 2002;26(2):89–111.
[40] Garcia X, Latham J, Xiang J, Harrison J. A clustered overlapping sphere algorithm to represent real particles in discrete element modelling. Geotechnique 2009;59(9):779–84.
[41] Akgiray Ö, Saatçı AM. A new look at filter backwash hydraulics. Water Sci Technol Water Supply 2001;1(2):65–72.
[42] Schroeder WJ, Lorensen B, Martin K. The visualization toolkit: An object-oriented approach to 3D graphics. Kitware; 2004.
[43] Tong Z, Zheng B, Yang R, Yu A, Chan H. CFD-DEM investigation of the dispersion mechanisms in commercial dry powder inhalers. Powder Technol 2013;240:19–24.
[44] Jasak H, Jemcov A, Tukovic Z, et al. OpenFOAM: A C++ library for complex physics simulations. In: International workshop on coupled methods in numerical dynamics, vol. 1000. Croatia: IUC Dubrovnik; 2007. p. 1–20.
[45] Abi-Mansour A, McClure S, Gentzler M. XRCT characterization of mesoscopic structure in poured and tapped cohesive powders and prediction by DEM. Powder Technol 2018;330:386–96.