sphere Documentation

Release 1.00-alpha

Anders Damsgaard

March 19, 2014

CONTENTS

This is the official documentation for the `sphere` discrete element modelling software. This document aims at guiding the installation process, documenting the usage, and explaining the relevant theory.

`sphere` is developed by Anders Damsgaard as part as his Ph.D. project, under supervision of David Lundbek Egholm and Jan A. Piotrowski, all of the Department of Geoscience, Aarhus University, Denmark. The author welcomes interested third party developers. This document is a work in progress.

Contact: Anders Damsgaard, http://cs.au.dk/~adc, mailto:anders.damsgaard@geo.au.dk

CHAPTER

ONE

CONTENTS

1.1 Introduction

The `sphere`-software is used for three-dimensional discrete element method (DEM) particle simulations. The source code is written in C++, CUDA C and Python, and is compiled by the user. The main computations are performed on the graphics processing unit (GPU) using NVIDIA's general purpose parallel computing architecture, CUDA. Simulation setup and data analysis is performed with the included Python API.

The ultimate aim of the `sphere` software is to simulate soft-bedded subglacial conditions, while retaining the flexibility to perform simulations of granular material in other environments.

The purpose of this documentation is to provide the user with a walk-through of the installation, work-flow, data-analysis and visualization methods of `sphere`. In addition, the `sphere` internals are exposed to provide a way of understanding of the discrete element method numerical routines taking place.

Note: Command examples in this document starting with the symbol `$` are meant to be executed in the shell of the operational system, and `>>>` means execution in Python. IPython is an excellent, interactive Python shell.

All numerical values in this document, the source code, and the configuration files are typeset with strict respect to the SI unit system.

1.1.1 Requirements

The build requirements are:

- A Nvidia CUDA-supported version of Linux or Mac OS X (see the CUDA toolkit release notes for more information)
- GNU Make
- CMake, version 2.8 or newer
- The GNU Compiler Collection (GCC)
- The Nvidia CUDA toolkit, version 5.0 or newer

In Debian GNU/Linux, these dependencies can be installed by running:

```
$ sudo apt-get install build-essential cmake nvidia-cuda-toolkit
```

Unfortunately, the Nvidia Toolkit is shipped under a non-free license. In order to install it in Debian GNU/Linux, add `non-free` archives to your `/etc/apt/sources.list`.

The runtime requirements are:

- A CUDA-enabled GPU with compute capability 1.1 or greater.
- A Nvidia CUDA-enabled GPU and device driver

Optional tools, required for simulation setup and data processing:

- Python
- Numpy
- Matplotlib
- Python bindings for VTK
- Imagemagick
- ffmpeg. Soon to be replaced by avconv!

In Debian GNU/Linux, these dependencies can be installed by running:

```
$ sudo apt-get install python python-numpy python-matplotlib python-vtk \
    imagemagick libav-tools
```

`sphere` is distributed with a HTML and PDF build of the documentation. The following tools are required for building the documentation:

- Sphinx
  - sphinxcontrib-programoutput
- Doxygen
- Breathe
- dvipng
- TeX Live, including `pdflatex`

In Debian GNU/Linux, these dependencies can be installed by running:

```
$ sudo apt-get install python-sphinx python-pip doxygen dvipng \
    python-sphinxcontrib-programoutput texlive-full
$ sudo pip install breathe
```

Git is used as the distributed version control system platform, and the source code is maintained at Github. `sphere` is licensed under the GNU Public License, v.3.

Note: All Debian GNU/Linux runtime, optional, and documentation dependencies mentioned above can be installed by executing the following command from the `doc/` folder:

```
$ make install-debian-pkgs
```

1.1.2 Obtaining sphere

The best way to keep up to date with subsequent updates, bugfixes and development, is to use the Git version control system. To obtain a local copy, execute:

```
$ git clone git@github.com:anders-dc/sphere.git
```

1.1.3 Building `sphere`

`sphere` is built using `cmake`, the platform-specific C/C++ compilers, and `nvcc` from the Nvidia CUDA toolkit.

If you plan to run `sphere` on a Kepler GPU, execute the following commands from the root directory:

```
$ cmake . && make
```

If you instead plan to execute it on a Fermi GPU, change `set(GPU_GENERATION 1)` to `set(GPU_GENERATION 0` in `CMakeLists.txt`.

In some cases the CMake FindCUDA module will have troubles locating the CUDA samples directory, and will complain about `helper_math.h` not being found.

In that case, modify the `CUDA_SDK_ROOT_DIR` variable in `src/CMakeLists.txt` to the path where you installed the CUDA samples, and run `cmake . && make` again. Alternatively, copy `helper_math.h` from the CUDA sample subdirectory `common/inc/helper_math.h` into the sphere `src/` directory, and run `cmake` and `make` again. Due to license restrictions, sphere cannot be distributed with this file.

After a successfull installation, the `sphere` executable will be located in the root folder. To make sure that all components are working correctly, execute:

```
$ make test
```

All instructions required for building `sphere` are provided in a number of `Makefile`'s. To generate the main `sphere` command-line executable, go to the root directory, and invoke CMake and GNU Make:

```
$ cmake . && make
```

If successfull, the Makefiles will create the required data folders, object files, as well as the `sphere` executable in the root folder. Issue the following commands to check the executable:

```
$ ./sphere --version
```

The output should look similar to this:

```
.--------------------------------------.
|               _    Compiled for 3D   |
|              | |                     |
|    ___ _ __  | |__   ___ _ __ ___    |
|   / __| '_ \| '_ \ / _ \ '__/ _ \   |
|   \__ \ |_) | | | |  __/ | |  __/   |
|   |___/ .__/|_| |_|\___|_|  \___|   |
|       | |                           |
|       |_|           Version: 0.35    |
`--------------------------------------´
 A discrete element method particle dynamics simulator.
 Written by Anders Damsgaard Christensen, license GPLv3+.
```

The build can be verified by running a number of automated tests:

```
$ make test
```

The documentation can be read in the reStructuredText-format in the `doc/sphinx/` folder, or in the HTML or PDF formats in the folders `doc/html` and `doc/pdf`.

Optionally, the documentation can be built using the following commands:

```
$ cd doc/sphinx
$ make html
$ make latexpdf
```

To see all available output formats, execute:

```
$ make help
```

1.1.4 Updating sphere

To update your local version, type the following commands in the `sphere` root directory:

```
$ git pull && cmake . && make
```

1.1.5 Work flow

After compiling the `sphere` binary, the procedure of a creating and handling a simulation is typically arranged in the following order:

- Setup of particle assemblage, physical properties and conditions using the Python API.
- Execution of `sphere` software, which simulates the particle behavior as a function of time, as a result of the conditions initially specified in the input file.
- Inspection, analysis, interpretation and visualization of `sphere` output in Python, and/or scene rendering using the built-in ray tracer.

1.2 Discrete element method

Granular material is a very common form of matter, both in nature and industry. It can be defined as material consisting of interacting, discrete particles. Common granular materials include gravels, sands and soils, ice bergs, asteroids, powders, seeds, and other foods. Over 75% of the raw materials that pass through industry are granular. This wide occurrence has driven the desire to understand the fundamental mechanics of the material.

Contrary to other common materials such as gases, liquids and solids, a general mathematical formulation of it's behavior hasn't yet been found. Granular material can, however, display states that somewhat resemble gases, fluids and solids.

The Discrete Element Method (DEM) is a numerical method that can be used to simulate the interaction of particles. Originally derived from Molecular Dynamics, it simulates particles as separate entities, and calculates their positions, velocities, and accelerations through time. See Cundall and Strack (1979) and this blog post for general introduction to the DEM. The following sections will highlight the DEM implementation in `sphere`. Some of the details are also described in Damsgaard et al. 2013. In the used notation, a bold symbol denotes a three-dimensional vector, and a dot denotes that the entity is a temporal derivative.

1.2.1 Contact search

Homogeneous cubic grid.

$$\delta_n^{ij} = ||\boldsymbol{x}^i - \boldsymbol{x}^j|| - (r^i + r^j)$$

where r is the particle radius, and $\boldsymbol{x}$ denotes the positional vector of a particle, and i and j denote the indexes of two particles. Negative values of δ_n denote that the particles are overlapping.

1.2.2 Contact interaction

Now that the inter-particle contacts have been identified and characterized by their overlap, the resulting forces from the interaction can be resolved. The interaction is decomposed into normal and tangential components, relative to the contact interface orientation. The normal vector to the contact interface is found by:

$$\boldsymbol{n}^{ij} = \frac{\boldsymbol{x}^i - \boldsymbol{x}^j}{||\boldsymbol{x}^i - \boldsymbol{x}^j||}$$

The contact velocity $\dot{\boldsymbol{\delta}}$ is found by:

$$\dot{\boldsymbol{\delta}}^{ij} = (\boldsymbol{x}^i - \boldsymbol{x}^j) + (r^i + \frac{\delta_n^{ij}}{2})(\boldsymbol{n}^{ij} \times \boldsymbol{\omega}^i) + (r^j + \frac{\delta_n^{ij}}{2})(\boldsymbol{n}^{ij} \times \boldsymbol{\omega}^j)$$

The contact velocity is decomposed into normal and tangential components, relative to the contact interface. The normal component is:

$$\dot{\delta}_n^{ij} = -(\dot{\boldsymbol{\delta}}^{ij} \cdot \boldsymbol{n}^{ij})$$

and the tangential velocity component is found as:

$$\dot{\boldsymbol{\delta}}_t^{ij} = \dot{\boldsymbol{\delta}}^{ij} - \boldsymbol{n}^{ij}(\boldsymbol{n}^{ij} \cdot \dot{\boldsymbol{\delta}}^{ij})$$

where $\boldsymbol{\omega}$ is the rotational velocity vector of a particle. The total tangential displacement on the contact plane is found incrementally:

$$\boldsymbol{\delta}_{t,\text{uncorrected}}^{ij} = \int_0^{t_c} \dot{\boldsymbol{\delta}}_t^{ij} \Delta t$$

where t_c is the duration of the contact and Δt is the computational time step length. The tangential contact interface displacement is set to zero when a contact pair no longer overlaps. At each time step, the value of $\boldsymbol{\delta}_t$ is corrected for rotation of the contact interface:

$$\boldsymbol{\delta}_t^{ij} = \boldsymbol{\delta}_{t,\text{uncorrected}}^{ij} - (\boldsymbol{n}(\boldsymbol{n} \cdot \boldsymbol{\delta}_{t,\text{uncorrected}}^{ij})$$

With all the geometrical and kinetic components determined, the resulting forces of the particle interaction can be determined using a contact model. `sphere` features only one contact model in the normal direction to the contact; the linear-elastic-viscous (*Hookean* with viscous damping, or *Kelvin-Voigt*) contact model. The resulting force in the normal direction of the contact interface on particle i is:

$$\boldsymbol{f}_n^{ij} = \left(-k_n \delta_n^{ij} - \gamma_n \dot{\delta_n}^{ij}\right) \boldsymbol{n}^{ij}$$

The parameter k_n is the defined spring coefficient in the normal direction of the contact interface, and γ_n is the defined contact interface viscosity, also in the normal direction. The loss of energy in this interaction due to the viscous component is for particle i calculated as:

$$\dot{e}_v^i = \gamma_n (\dot{\delta}_n^{ij})^2$$

The tangential force is determined by either a viscous-frictional contact model, or a elastic-viscous-frictional contact model. The former contact model is very computationally efficient, but somewhat inaccurate relative to the mechanics of real materials. The latter contact model is therefore the default, even though it results in longer computational times. The tangential force in the visco-frictional contact model:

$$\boldsymbol{f}_t^{ij} = -\gamma_t \dot{\boldsymbol{\delta}}_t^{ij}$$

γ_n is the defined contact interface viscosity in the tangential direction. The tangential displacement along the contact interface ($\boldsymbol{\delta}_t$) is not calculated and stored for this contact model. The tangential force in the more realistic elastic-viscous-frictional contact model:

$$\boldsymbol{f}_t^{ij} = -k_t \boldsymbol{\delta}_t^{ij} - \gamma_t \dot{\boldsymbol{\delta}}_t^{ij}$$

The parameter k_n is the defined spring coefficient in the tangential direction of the contact interface. Note that the tangential force is only found if the tangential displacement (δ_t) or the tangential velocity ($\dot{\delta}_t$) is non-zero, in order to avoid division by zero. Otherwise it is defined as being $[0, 0, 0]$.

For both types of contact model, the tangential force is limited by the Coulomb criterion of static and dynamic friction:

$$||\boldsymbol{f}_t^{ij}|| \leq \begin{cases} \mu_s ||\boldsymbol{f}_n^{ij}|| & \text{if} \quad ||\boldsymbol{f}_t^{ij}|| = 0 \\ \mu_d ||\boldsymbol{f}_n^{ij}|| & \text{if} \quad ||\boldsymbol{f}_t^{ij}|| > 0 \end{cases}$$

If the elastic-viscous-frictional contact model is used and the Coulomb limit is reached, the tangential displacement along the contact interface is limited to this value:

$$\boldsymbol{\delta}_t^{ij} = \frac{1}{k_t} \left(\mu_d ||\boldsymbol{f}_n^{ij}|| \frac{\boldsymbol{f}_t^{ij}}{||\boldsymbol{f}_t^{ij}||} + \gamma_t \dot{\boldsymbol{\delta}}_t^{ij} \right)$$

If the tangential force reaches the Coulomb limit, the energy lost due to frictional dissipation is calculated as:

$$\dot{e}_s^i = \frac{||\boldsymbol{f}_t^{ij} \dot{\boldsymbol{\delta}}_t^{ij} \Delta t||}{\Delta t}$$

The loss of energy by viscous dissipation in the tangential direction is not found.

1.2.3 Temporal integration

In the DEM, the time is discretized into small steps (Δt). For each time step, the entire network of contacts is resolved, and the resulting forces and torques for each particle are found. With these values at hand, the new linear and rotational accelerations can be found using Newton's second law of the motion of solid bodies. If a particle with mass m at a point in time experiences a sum of forces denoted $\boldsymbol{F}$, the resultant acceleration ($\boldsymbol{a}$) can be found by rearranging Newton's second law:

$$\boldsymbol{F} = m\boldsymbol{a} \Rightarrow \boldsymbol{a} = \frac{\boldsymbol{F}}{m}$$

The new velocity and position is found by integrating the above equation with regards to time. The simplest integration scheme in this regard is the Euler method:

$$\boldsymbol{v} = \boldsymbol{v}_{old} + \boldsymbol{a}\Delta t$$

$$\boldsymbol{p} = \boldsymbol{p}_{old} + \boldsymbol{v}\Delta t$$

1.3 Fluid simulation and particle-fluid interaction

A new and experimental addition to *sphere* is the ability to simulate a mixture of particles and a Newtonian fluid. The fluid is simulated using an Eulerian continuum approach, using a custom CUDA solver for GPU computation. This approach allows for fast simulations due to the limited need for GPU-CPU communications, as well as a flexible code base.

The following sections will describe the theoretical background, as well as the solution procedure and the numerical implementation.

1.3.1 Derivation of the Navier Stokes equations with porosity

Following the outline presented by Limache and Idelsohn (2006), the continuity equation for an incompressible fluid material is given by:

$$\nabla \cdot \boldsymbol{v} = 0$$

and the momentum equation:

$$\rho \frac{\partial \boldsymbol{v}}{\partial t} + \rho(\boldsymbol{v} \cdot \nabla \boldsymbol{v}) = \nabla \cdot \boldsymbol{\sigma} + \rho \boldsymbol{f}$$

Here, $\boldsymbol{v}$ is the fluid velocity, ρ is the fluid density, $\boldsymbol{\sigma}$ is the Cauchy stress tensor, and $\boldsymbol{f}$ is a body force (e.g. gravity). For incompressible Newtonian fluids, the Cauchy stress is given by:

$$\boldsymbol{\sigma} = -p\boldsymbol{I} + \boldsymbol{\tau}$$

p is the fluid pressure, $\boldsymbol{I}$ is the identity tensor, and $\boldsymbol{\tau}$ is the deviatoric stress tensor, given by:

$$\boldsymbol{\tau} = \mu_f \nabla \boldsymbol{v} + \mu_f (\nabla \boldsymbol{v})^T$$

By using the following vector identities:

$$\nabla \cdot (p\boldsymbol{I}) = \nabla p$$
$$\nabla \cdot (\nabla \boldsymbol{v}) = \nabla^2 \boldsymbol{v}$$
$$\nabla \cdot (\nabla \boldsymbol{v})^T = \nabla(\nabla \cdot \boldsymbol{v})$$

the deviatoric component of the Cauchy stress tensor simplifies to the following, assuming that spatial variations in the viscosity can be neglected:

$$= -\nabla p + \mu_f \nabla^2 \boldsymbol{v}$$

Since we are dealing with fluid flow in a porous medium, additional terms are introduced to the equations for conservation of mass and momentum. In the following, the equations are derived for the first spatial component. The solution for the other components is trivial.

The porosity value (in the saturated porous medium the volumetric fraction of the fluid phase) denoted ϕ is incorporated in the continuity and momentum equations. The continuity equation becomes:

$$\frac{\partial \phi}{\partial t} + \nabla \cdot (\phi \boldsymbol{v}) = 0$$

For the x component, the Lagrangian formulation of the momentum equation with a body force $\boldsymbol{f}$ becomes:

$$\frac{D(\phi v_x)}{Dt} = \frac{1}{\rho} \left[\nabla \cdot (\phi \boldsymbol{\sigma})\right]_x + \phi f_x$$

In the Eulerian formulation, an advection term is added, and the Cauchy stress tensor is represented as isotropic and deviatoric components individually:

$$\frac{\partial (\phi v_x)}{\partial t} + \boldsymbol{v} \cdot \nabla (\phi v_x) = \frac{1}{\rho} \left[\nabla \cdot (-\phi p \boldsymbol{I}) + \phi \boldsymbol{\tau})\right]_x + \phi f_x$$

Using vector identities to rewrite the advection term, and expanding the fluid stress tensor term:

$$\frac{\partial (\phi v_x)}{\partial t} + \nabla \cdot (\phi v_x \boldsymbol{v}) - \phi v_x (\nabla \cdot \boldsymbol{v}) = \frac{1}{\rho} \left[-\nabla \phi p\right]_x + \frac{1}{\rho} \left[-\phi \nabla p\right]_x + \frac{1}{\rho} \left[\nabla \cdot (\phi \boldsymbol{\tau})\right]_x + \phi f_x$$

Spatial variations in the porosity are neglected,

$$\nabla \phi := 0$$

and the pressure is attributed to the fluid phase alone (model B in Zhu et al. 2007 and Zhou et al. 2010). The divergence of fluid velocities is defined to be zero:

$$\nabla \cdot \boldsymbol{v} := 0$$

With these assumptions, the momentum equation simplifies to:

$$\frac{\partial (\phi v_x)}{\partial t} + \nabla \cdot (\phi v_x \boldsymbol{v}) = -\frac{1}{\rho} \frac{\partial p}{\partial x} + \frac{1}{\rho} \left[\nabla \cdot (\phi \boldsymbol{\tau})\right]_x + \phi f_x$$

The remaining part of the advection term is for the x component found as:

$$\nabla \cdot (\phi v_x \boldsymbol{v}) = \left[\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right] \begin{bmatrix} \phi v_x v_x \\ \phi v_x v_y \\ \phi v_x v_z \end{bmatrix} = \frac{\partial (\phi v_x v_x)}{\partial x} + \frac{\partial (\phi v_x v_y)}{\partial y} + \frac{\partial (\phi v_x v_z)}{\partial z}$$

The deviatoric stress tensor is in this case symmetrical, i.e. $\tau_{ij} = \tau_{ji}$, and is found by:

$$\frac{1}{\rho}\left[\nabla \cdot (\phi\boldsymbol{\tau})\right]_x = \frac{1}{\rho}\left[\left[\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z}\right]\phi\begin{bmatrix}\tau_{xx} & \tau_{xy} & \tau_{xz}\\ \tau_{yx} & \tau_{yy} & \tau_{yz}\\ \tau_{zx} & \tau_{zy} & \tau_{zz}\end{bmatrix}\right]_x$$

$$= \frac{1}{\rho}\begin{bmatrix}\frac{\partial(\phi\tau_{xx})}{\partial x} + \frac{\partial(\phi\tau_{xy})}{\partial y} + \frac{\partial(\phi\tau_{xz})}{\partial z}\\ \frac{\partial(\phi\tau_{yx})}{\partial x} + \frac{\partial(\phi\tau_{yy})}{\partial y} + \frac{\partial(\phi\tau_{yz})}{\partial z}\\ \frac{\partial(\phi\tau_{zx})}{\partial x} + \frac{\partial(\phi\tau_{zy})}{\partial y} + \frac{\partial(\phi\tau_{zz})}{\partial z}\end{bmatrix}_x = \frac{1}{\rho}\left(\frac{\partial(\phi\tau_{xx})}{\partial x} + \frac{\partial(\phi\tau_{xy})}{\partial y} + \frac{\partial(\phi\tau_{xz})}{\partial z}\right)$$

In a linear viscous fluid, the stress and strain rate ($\dot{\epsilon}$) is linearly dependent, scaled by the viscosity parameter μ_f:

$$\tau_{ij} = 2\mu_f\dot{\epsilon}_{ij} = \mu_f\left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}\right)$$

With this relationship, the deviatoric stress tensor components can be calculated as:

$$\tau_{xx} = 2\mu_f\frac{\partial v_x}{\partial x} \qquad \tau_{yy} = 2\mu_f\frac{\partial v_y}{\partial y} \qquad \tau_{zz} = 2\mu_f\frac{\partial v_z}{\partial z}$$

$$\tau_{xy} = \mu_f\left(\frac{\partial v_x}{\partial y} + \frac{\partial v_y}{\partial x}\right)$$

$$\tau_{xz} = \mu_f\left(\frac{\partial v_x}{\partial z} + \frac{\partial v_z}{\partial x}\right)$$

$$\tau_{yz} = \mu_f\left(\frac{\partial v_y}{\partial z} + \frac{\partial v_z}{\partial y}\right)$$

where μ_f is the dynamic viscosity. The above formulation of the fluid rheology assumes identical bulk and shear viscosities. The derivation of the equations for the other spatial components is trivial.

1.3.2 Porosity estimation

The solid volume in each fluid cell is determined by the ratio of the a cell-centered spherical cell volume (V_c) and the sum of intersecting particle volumes (V_s). The spherical cell volume has a center at $\boldsymbol{x}_i$, and a radius of R_i, which is equal to half the fluid cell width. The nearby particles are characterized by position $\boldsymbol{x}_j$ and radius r_j. The center distance is defined as:

$$d_{ij} = ||\boldsymbol{x}_i - \boldsymbol{x}_j||$$

The common volume of the two intersecting spheres is zero if the volumes aren't intersecting, lens shaped if they are intersecting, and spherical if the particle is fully contained by the spherical cell volume:

$$V_i^s = \sum_j \begin{cases} 0 & \text{if } R_i + r_j \leq d_{ij} \\ \frac{1}{12d_{ij}}\left[\pi(R_i + r_j - d_{ij})^2(d_{ij}^2 + 2d_{ij}r_j - 3r_j^2 + 2d_{ij}R_i + 6r_jR_i - 3R_i^2)\right] & \text{if } R_i - r_j < d_{ij} < R_i + r_j \\ \frac{4}{3}\pi r_j^3 & \text{if } d_{ij} \leq R_i - r_j \end{cases}$$

Using this method, the cell porosity values are continuous through time as particles enter and exit the cell volume. The rate of porosity change ($d\phi/dt$) is estimated by the backwards Euler method by considering the previous and current porosity.

1.3.3 Particle-fluid interaction

The momentum exchange of the granular and fluid phases follows the procedure outlined by Gidaspow 1992 and Shamy and Zhegal 2005. The fluid and particle interaction is based on the concept of drag, where the magnitude is based on semi-empirical relationships. The drag force scales linearly with the relative difference in velocity between the fluid and particle phase. On the base of Newton's third law, the resulting drag force is applied with opposite signs to the particle and fluid.

For fluid cells with porosities (ϕ) less or equal to 0.8, the drag force is based on the Ergun (1952) equation:

$$\bar{\boldsymbol{f}}_d = \left(150\frac{\mu_f(1-\phi)^2}{\phi\bar{d}^2} + 1.75\frac{(1-\phi)\rho_f||\boldsymbol{v}_f - \bar{\boldsymbol{v}}_p||}{\bar{d}}\right)(\boldsymbol{v}_f - \bar{\boldsymbol{v}}_p)$$

here, $\bar{d}$ denotes the average particle diameter in the cell, $\boldsymbol{v}_f$ is the fluid flow velocity, and $\bar{\boldsymbol{v}}_p$ is the average particle velocity in the cell. All particles in contact with the previously mentioned cell-centered sphere for porosity estimation contribute to the average particle velocity and diameter in the fluid cell.

If the porosity is greater than 0.8, the cell-averaged drag force ($\bar{\boldsymbol{f}}_d$ is found from the Wen and Yu (1966) equation, which considers the fluid flow situation:

$$\bar{\boldsymbol{f}}_d = \left(\frac{3}{4}\frac{C_d(1-\phi)\phi^{-2.65}\mu_f\rho_f||\boldsymbol{v}_f - \bar{\boldsymbol{v}}_p||}{\bar{d}}\right)(\boldsymbol{v}_f - \bar{\boldsymbol{v}}_p)$$

The drag coefficient C_d is evaluated depending on the magnitude of the Reynolds number Re:

$$C_d = \begin{cases} \frac{24}{Re}(1 + 0.15(Re)^{0.687} & \textit{if } Re < 1,000 \\ 0.44 & \textit{if } Re \geq 1,000 \end{cases}$$

where the Reynold's number is found by:

$$Re = \frac{\phi\rho_f\bar{d}}{\mu_f}||\boldsymbol{v}_f - \bar{\boldsymbol{v}}_p||$$

The interaction force is applied to the fluid with negative sign as a contribution to the body force $\boldsymbol{f}$. The fluid interaction force applied particles in the fluid cell is:

$$\boldsymbol{f}_i = \frac{\bar{\boldsymbol{f}}_d V_p}{1-\phi}$$

where V_p denotes the particle volume. Optionally, the above interaction force could be expanded to include the force induced by the fluid pressure gradient:

$$\boldsymbol{f}_i = \left(-\nabla p + \frac{\bar{\boldsymbol{f}}_d}{1-\phi}\right)V_p$$

1.3.4 Fluid dynamics solution procedure by operator splitting

The partial differential terms in the previously described equations are found using finite central differences. Modifying the operator splitting methodology presented by Langtangen et al. (2002), the predicted velocity $\boldsymbol{v}^*$ after a finite

time step Δt is found by explicit integration of the momentum equation.

$$\frac{\Delta(\phi v_x)}{\Delta t} + \nabla \cdot (\phi v_x \boldsymbol{v}) = -\frac{1}{\rho}\frac{\Delta p}{\Delta x} + \frac{1}{\rho}\left[\nabla \cdot (\phi \boldsymbol{\tau})\right]_x + \phi f_x$$
$$\Downarrow$$
$$\phi\frac{\Delta v_x}{\Delta t} + v_x\frac{\Delta \phi}{\Delta t} + \nabla \cdot (\phi v_x \boldsymbol{v}) = -\frac{1}{\rho}\frac{\Delta p}{\Delta x} + \frac{1}{\rho}\left[\nabla \cdot (\phi \boldsymbol{\tau})\right]_x + \phi f_x$$

We want to isolate Δv_x in the above equation in order to project the new velocity.

$$\phi\frac{\Delta v_x}{\Delta t} = -\frac{1}{\rho}\frac{\Delta p}{\Delta x} + \frac{1}{\rho}\left[\nabla \cdot (\phi \boldsymbol{\tau})\right]_x + \phi f_x - v_x\frac{\Delta \phi}{\Delta t} - \nabla \cdot (\phi v_x \boldsymbol{v})$$
$$\Delta v_x = -\frac{1}{\rho}\frac{\Delta p}{\Delta x}\frac{\Delta t}{\phi} + \frac{1}{\rho}\left[\nabla \cdot (\phi \boldsymbol{\tau})\right]_x \frac{\Delta t}{\phi} + \Delta t f_x - v_x\frac{\Delta \phi}{\phi} - \nabla \cdot (\phi v_x \boldsymbol{v})\frac{\Delta t}{\phi}$$

The term β is introduced as an adjustable, dimensionless parameter in the range $[0; 1]$, and determines the importance of the old pressure values in the solution procedure (Langtangen et al. 2002). A value of 0 corresponds to Chorin's projection method originally described in Chorin (1968).

$$v_x^* = v_x^t + \Delta v_x$$
$$v_x^* = v_x^t - \frac{\beta}{\rho}\frac{\Delta p^t}{\Delta x}\frac{\Delta t}{\phi^t} + \frac{1}{\rho}\left[\nabla \cdot (\phi^t \boldsymbol{\tau}^t)\right]_x \frac{\Delta t}{\phi} + \Delta t f_x - v_x^t\frac{\Delta \phi}{\phi^t} - \nabla \cdot (\phi^t v_x^t \boldsymbol{v}^t)\frac{\Delta t}{\phi^t}$$

Here, Δx denotes the cell spacing. The velocity found (v_x^*) is only a prediction of the fluid velocity at time $t + \Delta t$, since the estimate isn't constrained by the continuity equation:

$$\frac{\Delta \phi^t}{\Delta t} + \nabla \cdot (\phi^t \boldsymbol{v}^{t+\Delta t}) = 0$$

The divergence of a scalar and vector can be split:

$$\phi^t \nabla \cdot \boldsymbol{v}^{t+\Delta t} + \boldsymbol{v}^{t+\Delta t} \cdot \nabla \phi^t + \frac{\Delta \phi^t}{\Delta t} = 0$$

The predicted velocity is corrected using the new pressure (Langtangen et al. 2002):

$$\boldsymbol{v}^{t+\Delta t} = \boldsymbol{v}^* - \frac{\Delta t}{\rho}\nabla \epsilon \quad \text{where} \quad \epsilon = p^{t+\Delta t} - \beta p^t$$

The above formulation of the future velocity is put into the continuity equation:

$$\Rightarrow \phi^t \nabla \cdot \left(\boldsymbol{v}^* - \frac{\Delta t}{\rho}\nabla \epsilon\right) + \left(\boldsymbol{v}^* - \frac{\Delta t}{\rho}\nabla \epsilon\right) \cdot \nabla \phi^t + \frac{\Delta \phi^t}{\Delta t} = 0$$

$$\Rightarrow \phi^t \nabla \cdot \boldsymbol{v}^* - \frac{\Delta t}{\rho}\phi^t \nabla^2 \epsilon + \nabla \phi^t \cdot \boldsymbol{v}^* - \nabla \phi^t \cdot \nabla \epsilon \frac{\Delta t}{\rho} + \frac{\Delta \phi^t}{\Delta t} = 0$$

$$\Rightarrow \frac{\Delta t}{\rho}\phi^t \nabla^2 \epsilon = \phi^t \nabla \cdot \boldsymbol{v}^* + \nabla \phi^t \cdot \boldsymbol{v}^* - \nabla \phi^t \cdot \nabla \epsilon \frac{\Delta t}{\rho} + \frac{\Delta \phi^t}{\Delta t}$$

The pressure difference in time becomes a Poisson equation with added terms:

$$\Rightarrow \nabla^2 \epsilon = \frac{\nabla \cdot \boldsymbol{v}^* \rho}{\Delta t} + \frac{\nabla \phi^t \cdot \boldsymbol{v}^* \rho}{\Delta t \phi^t} - \frac{\nabla \phi^t \cdot \nabla \epsilon}{\phi^t} + \frac{\Delta \phi^t \rho}{\Delta t^2 \phi^t}$$

The right hand side of the above equation is termed the *forcing function* f, which is decomposed into two terms, f_1 and f_2:

$$f_1 = \frac{\nabla \cdot \boldsymbol{v}^* \rho}{\Delta t} + \frac{\nabla \phi^t \cdot \boldsymbol{v}^* \rho}{\Delta t \phi^t} + \frac{\Delta \phi^t \rho}{\Delta t^2 \phi^t}$$

$$f_2 = \frac{\nabla \phi^t \cdot \nabla \epsilon}{\phi^t}$$

During the Jacobi iterative solution procedure f_1 remains constant, while f_2 changes value. For this reason, f_1 is found only during the first iteration, while f_2 is updated every time. The value of the forcing function is found as:

$$f = f_1 - f_2$$

Using second-order finite difference approximations of the Laplace operator second-order partial derivatives, the differential equations become a system of equations that is solved using iteratively using Jacobi updates. The total number of unknowns is $(n_x - 1)(n_y - 1)(n_z - 1)$.

The discrete Laplacian (approximation of the Laplace operator) can be obtained by a finite-difference seven-point stencil in a three-dimensional, cubic grid with cell spacing $\Delta x, \Delta y, \Delta z$, considering the six face neighbors:

$$\nabla^2 \epsilon_{i_x,i_y,i_z} \approx \frac{\epsilon_{i_x-1,i_y,i_z} - 2\epsilon_{i_x,i_y,i_z} + \epsilon_{i_x+1,i_y,i_z}}{\Delta x^2} + \frac{\epsilon_{i_x,i_y-1,i_z} - 2\epsilon_{i_x,i_y,i_z} + \epsilon_{i_x,i_y+1,i_z}}{\Delta y^2}$$
$$+ \frac{\epsilon_{i_x,i_y,i_z-1} - 2\epsilon_{i_x,i_y,i_z} + \epsilon_{i_x,i_y,i_z+1}}{\Delta z^2} \approx f_{i_x,i_y,i_z}$$

Within a Jacobi iteration, the value of the unknowns (ϵ^n) is used to find an updated solution estimate (ϵ^{n+1}). The solution for the updated value takes the form:

$$\epsilon^{n+1}_{i_x,i_y,i_z} = \frac{-\Delta x^2 \Delta y^2 \Delta z^2 f_{i_x,i_y,i_z} + \Delta y^2 \Delta z^2 (\epsilon^n_{i_x-1,i_y,i_z} + \epsilon^n_{i_x+1,i_y,i_z}) + \Delta x^2 \Delta z^2 (\epsilon^n_{i_x,i_y-1,i_z} + \epsilon^n_{i_x,i_y+1,i_z}) + \Delta x^2 \Delta y^2 (\epsilon^n_{i_x,i_y,i_z-1} +}{2(\Delta x^2 \Delta y^2 + \Delta x^2 \Delta z^2 + \Delta y^2 \Delta z^2)}$$

The difference between the current and updated value is termed the *normalized residual*:

$$r_{i_x,i_y,i_z} = \frac{(\epsilon^{n+1}_{i_x,i_y,i_z} - \epsilon^n_{i_x,i_y,i_z})^2}{(\epsilon^{n+1}_{i_x,i_y,i_z})^2}$$

Note that the ϵ values cannot be 0 due to the above normalization of the residual.

The updated values are at the end of the iteration stored as the current values, and the maximal value of the normalized residual is found. If this value is larger than a tolerance criteria, the procedure is repeated. The iterative procedure is ended if the number of iterations exceeds a defined limit.

After the values of ϵ are found, they are used to find the new pressures and velocities:

$$\bar{p}^{t+\Delta t} = \beta \bar{p}^t + \epsilon$$

$$\bar{\boldsymbol{v}}^{t+\Delta t} = \bar{\boldsymbol{v}}^* - \frac{\Delta t}{\rho} \nabla \epsilon$$

1.3.5 Boundary conditions

The lateral boundaries are periodic. This cannot be changed in the current version of `sphere`. This means that the fluid properties at the paired, parallel lateral (x and y) boundaries are identical. A flow leaving through one side reappears on the opposite side.

The top and bottom boundary conditions of the fluid grid can be either: prescribed pressure (Dirichlet), or prescribed velocity (Neumann). The (horizontal) velocities parallel to the boundaries are free to attain other values (free slip). The Dirichlet boundary condition is enforced by keeping the value of ϵ constant at the boundaries, e.g.:

$$\epsilon^{n+1}_{i_x,i_y,i_z=1 \vee n_z} = \epsilon^{n}_{i_x,i_y,i_z=1 \vee n_z}$$

The Neumann boundary condition of no flow across the boundary is enforced by setting the gradient of ϵ perpendicular to the boundary to zero, e.g.:

$$\nabla_z \epsilon^{n+1}_{i_x,i_y,i_z=1 \vee n_z} = 0$$

1.3.6 Numerical implementation

Ghost nodes

—

1.4 Python API

The Python module `sphere` is intended as the main interface to the `sphere` application. It is recommended to use this module for simulation setup, simulation execution, and analysis of the simulation output data.

In order to use the API, the file `sphere.py` must be placed in the same directory as the Python files.

1.4.1 Sample usage

Below is a simple, annotated example of how to setup, execute, and post-process a `sphere` simulation. The example is also found in the `python/` folder as `collision.py`.

```
#!/usr/bin/env python
"""
Example of two particles colliding.
Place script in sphere/python/ folder, and invoke with 'python collision.py'
"""

# Import the sphere module for setting up, running, and analyzing the
# experiment. We also need the numpy module when setting arrays in the sphere
# object.
import sphere
import numpy


### SIMULATION SETUP

# Create a sphere object with two preallocated particles and a simulation ID
```

```
SB = sphere.sim(np = 2, sid = 'collision')

SB.radius[:] = 0.3 # set radii to 0.3 m

# Define the positions of the two particles
SB.x[0, :] = numpy.array([10.0, 5.0, 5.0])   # particle 1 (idx 0)
SB.x[1, :] = numpy.array([11.0, 5.0, 5.0])   # particle 2 (idx 1)

# The default velocity is [0,0,0]. Slam particle 1 into particle 2 by defining
# a positive x velocity for particle 1.
SB.vel[0, 0] = 1.0

# Set the world limits and the particle sorting grid. The particles need to stay
# within the world limits for the entire simulation, otherwise it will stop!
SB.initGridAndWorldsize(margin = 5.0)

# Define the temporal parameters, e.g. the total time (total) and the file
# output interval (file_dt), both in seconds
SB.initTemporal(total = 2.0, file_dt = 0.1)

# Using a 'dry' run, the sphere main program will display important parameters.
# sphere will end after displaying these values.
SB.run(dry = True)


### RUNNING THE SIMULATION

# Start the simulation on the GPU from the sphere program
SB.run()


### ANALYSIS OF SIMULATION RESULTS

# Plot the system energy through time, image saved as collision-energy.png
sphere.visualize(SB.sid, method = 'energy')

# Render the particles using the built-in raytracer
SB.render()

# Alternative visualization using ParaView. See the documentation of
#  ``sim.writeVTKall()`` for more information about displaying the
# particles in ParaView.
SB.writeVTKall()
```

The full documentation of the `sphere` Python API can be found below.

1.4.2 The `sphere` module

`sphere.`**`V_sphere`**(*r*)

Calculates the volume of a sphere with radius r

Returns The sphere volume [m^3]

Return type float

`sphere.`**`cleanup`**(*sim*)

Removes the input/output files and images belonging to the object simulation ID from the `input/`, `output/` and `img_out/` folders.

Parameters **spherebin** (*sim*) – A sim object

sphere.**convert**(*graphics_format='png'*, *folder='../img_out'*)

Converts all PPM images in img_out to graphics_format using Imagemagick. All PPM images are subsequently removed.

Parameters

- **graphics_format** (*str*) – Convert the images to this format
- **folder** (*str*) – The folder containing the PPM images to convert

sphere.**render**(*binary*, *method='pres'*, *max_val=1000.0*, *lower_cutoff=0.0*, *graphics_format='png'*, *verbose=True*)

Render target binary using the `sphere` raytracer.

Parameters

- **method** (*str*) – The color visualization method to use for the particles. Possible values are: 'normal': color all particles with the same color, 'pres': color by pressure, 'vel': color by translational velocity, 'angvel': color by rotational velocity, 'xdisp': color by total displacement along the x-axis, 'angpos': color by angular position.
- **max_val** (*float*) – The maximum value of the color bar
- **lower_cutoff** (*float*) – Do not render particles with a value below this value, of the field selected by `method`
- **graphics_format** (*str*) – Convert the PPM images generated by the ray tracer to this image format using Imagemagick
- **verbose** (*bool*) – Show verbose information during ray tracing

sphere.**run**(*binary*, *verbose=True*, *hideinputfile=False*)

Execute `sphere` with target binary file as input.

Parameters

- **binary** (*str*) – Input file for `sphere`
- **verbose** (*bool*) – Show `sphere` output
- **hideinputfile** (*bool*) – Hide the input file

class sphere.**sim**(*sid='unnamed'*, *np=0*, *nd=3*, *nw=0*, *fluid=False*)

Class containing all `sphere` data.

Contains functions for reading and writing binaries, as well as simulation setup and data analysis. Most arrays are initialized to default values.

Parameters

- **np** (*int*) – The number of particles to allocate memory for (default = 1)
- **nd** (*int*) – The number of spatial dimensions (default = 3). Note that 2D and 1D simulations currently are not possible.
- **nw** (*int*) – The number of dynamic walls (default = 1)
- **sid** (*str*) – The simulation id (default = 'unnamed'). The simulation files will be written with this base name.
- **fluid** (*bool*) – Setup fluid simulation (default = False)

acceleration(*idx=-1*)

Returns the acceleration of one or more particles, selected by their index.

Parameters **idx** (*int, list or numpy.array*) – Index or index range of particles

Returns n-by-3 matrix of acceleration(s)

Return type numpy.array

addParticle(*x*, *radius*, *xysum=array([0., 0.])*, *vel=array([0., 0., 0.])*, *fixvel=array([0.])*, *force=array([0., 0., 0.])*, *angpos=array([0., 0., 0.])*, *angvel=array([0., 0., 0.])*, *torque=array([0., 0., 0.])*, *es_dot=array([0.])*, *es=array([0.])*, *ev_dot=array([0.])*, *ev=array([0.])*, *p=array([0.])*)

Add a single particle to the simulation object. The only required parameters are the position (x) and the radius (radius).

Parameters

- **x** (*numpy.array*) – A vector pointing to the particle center coordinate.
- **radius** (*float*) – The particle radius
- **vel** (*numpy.array*) – The particle linear velocity (default = [0,0,0])
- **fixvel** (*float*) – Fix horizontal linear velocity (0: No, 1: Yes, default=0)
- **angpos** (*numpy.array*) – The particle angular position (default = [0,0,0])
- **angvel** (*numpy.array*) – The particle angular velocity (default = [0,0,0])
- **torque** (*numpy.array*) – The particle torque (default = [0,0,0])
- **es_dot** (*float*) – The particle shear energy loss rate (default = 0)
- **es** (*float*) – The particle shear energy loss (default = 0)
- **ev_dot** (*float*) – The particle viscous energy rate loss (default = 0)
- **ev** (*float*) – The particle viscous energy loss (default = 0)
- **p** (*float*) – The particle pressure (default = 0)

adjustUpperWall(*z_adjust=1.1*)

Included for legacy purposes, calls `adjustWall()` with `idx=0`.

Parameters **z_adjust** (*float*) – Increase the world and grid size by this amount to allow for wall movement.

adjustWall(*idx*, *adjust=1.1*)

Adjust grid and dynamic wall to max. particle position

Param idx: The wall to adjust. 0 = +z, upper wall (default), 1 = -x, left wall, 2 = +x, right wall, 3 = -y, front wall, 4 = +y, back wall.

Parameters **z_adjust** (*float*) – Increase the world and grid size by this amount to allow for wall movement.

bond(*i*, *j*)

Create a bond between particles with index i and j

Parameters

- **i** (*int*) – Index of first particle in bond
- **j** (*int*) – Index of second particle in bond

bondsRose(*graphics_format='pdf'*)

Visualize the trend and plunge angles of the bond pairs in a rose plot. The plot is saved in the current folder as 'bonds-<simulation id>-rose.<graphics_format>'.

Parameters **graphics_format** (*str*) – Save the plot in this format

bulkPorosity()
Calculates the bulk porosity

Returns The bulk porosity, in [0:1]

Return type float

consolidate(*normal_stress=10000.0*)
Setup consolidation experiment. Specify the upper wall normal stress in Pascal, default value is 10 kPa.

Parameters **normal_stress** (*float*) – The normal stress to apply from the upper wall

contactModel(*contactmodel*)
Define which contact model to use for the tangential component of particle-particle interactions. The elastic-viscous-frictional contact model (2) is considered to be the most realistic contact model, while the viscous-frictional contact model is significantly faster.

Parameters **contactmodel** (*int*) – The type of tangential contact model to use (visco-frictional = 1, elasto-visco-frictional = 2)

createBondPair(*i*, *j*, *spacing=-0.1*)
Bond particles i and j. Particle j is moved adjacent to particle i, and oriented randomly.

Parameters

- **i** (*int*) – Index of first particle in bond
- **j** (*int*) – Index of second particle in bond
- **spacing** (*float*) – The inter-particle distance prescribed. Positive values result in a inter-particle distance, negative equal an overlap. The value is relative to the sum of the two radii.

currentNormalStress()
Calculates the current magnitude of the top wall normal stress.

Returns The current top wall normal stress in Pascal

Return type float

defaultParams(*mu_s=0.4*, *mu_d=0.4*, *mu_r=0.0*, *rho=2600*, *k_n=1160000000.0*, *k_t=1160000000.0*, *k_r=0*, *gamma_n=0.0*, *gamma_t=0.0*, *gamma_r=0.0*, *gamma_wn=10000.0*, *gamma_wt=10000.0*, *capillaryCohesion=0*)
Initialize particle parameters to default values.

Parameters

- **mu_s** (*float*) – The coefficient of static friction between particles [-]
- **mu_d** (*float*) – The coefficient of dynamic friction between particles [-]
- **rho** (*float*) – The density of the particle material [kg/(m^3)]
- **k_n** (*float*) – The normal stiffness of the particles [N/m]
- **k_t** (*float*) – The tangential stiffness of the particles [N/m]
- **k_r** (*float*) – The rolling stiffness of the particles [N/rad] *Parameter not used*
- **gamma_n** (*float*) – Particle-particle contact normal viscosity [Ns/m]
- **gamma_t** (*float*) – Particle-particle contact tangential viscosity [Ns/m]
- **gamma_r** (*float*) – Particle-particle contact rolling viscosity *Parameter not used*
- **gamma_wn** (*float*) – Wall-particle contact normal viscosity [Ns/m]
- **gamma_wt** (*float*) – Wall-particle contact tangential viscosity [Ns/m]

- **capillaryCohesion** (*int*) – Enable particle-particle capillary cohesion interaction model (0 = no (default), 1 = yes)

defineWorldBoundaries(*L, origo=[0.0, 0.0, 0.0]*)

Set the boundaries of the world. Particles will only be able to interact within this domain. With dynamic walls, allow space for expansions. *Important*: The particle radii have to be set beforehand. The world edges act as static walls.

Parameters

- **L** (*numpy.array*) – The upper boundary of the domain [m]
- **origo** (*numpy.array*) – The lower boundary of the domain [m]. Negative values won't work. Default = [0.0, 0.0, 0.0].

disableFluidPressureModulation()

Set the parameters for the sine wave modulating the fluid pressures at the top boundary to zero.

See also: `setFluidPressureModulation()`

energy(*method*)

Calculates the sum of the energy components of all particles.

Parameters **method** (*str*) – The type of energy to return. Possible values are 'pot' for potential energy [J], 'kin' for kinetic energy [J], 'rot' for rotational energy [J], 'shear' for energy lost by friction, 'shearrate' for the rate of frictional energy loss [W], 'visc_n' for viscous losses normal to the contact [J], 'visc_n_rate' for the rate of viscous losses normal to the contact [W], and finally 'bondpot' for the potential energy stored in bonds [J]

Returns The value of the selected energy type

Return type float

forcechains(*lc=200.0*, *uc=650.0*, *outformat='png'*, *disp='2d'*)

Visualizes the force chains in the system from the magnitude of the normal contact forces, and produces an image of them. Warning: Will segfault if no contacts are found.

Parameters

- **lc** (*float*) – Lower cutoff of contact forces. Contacts below are not visualized
- **uc** (*float*) – Upper cutoff of contact forces. Contacts above are visualized with this value
- **outformat** (*str*) – Format of output image. Possible values are 'interactive', 'png', 'epslatex', 'epslatex-color'
- **disp** (*str*) – Display forcechains in '2d' or '3d'

forcechainsRose(*lower_limit=0.25*, *graphics_format='pdf'*)

Visualize trend and plunge angles of the strongest force chains in a rose plot. The plots are saved in the current folder with the name 'fc-<simulation id>-rose.pdf'.

Parameters

- **lower_limit** (*float*) – Do not visualize force chains below this relative contact force magnitude, in]0;1[
- **graphics_format** (*str*) – Save the plot in this format

generateBimodalRadii(*r_small=0.005*, *r_large=0.05*, *ratio=0.2*, *verbose=True*)

Draw random radii from two distinct sizes.

Parameters

- **r_small** (*float*) – Radii of small population [m], in]0;r_large[

- **r_large** (*float*) – Radii of large population [m], in]r_small;inf[
- **ratio** (*float*) – Approximate volumetric ratio between the two populations (large/small).

See also: `generateRadii()`.

generateRadii(*psd='logn'*, *radius_mean=0.00044*, *radius_variance=8.8e-09*, *histogram=True*)

Draw random particle radii from a selected probability distribution. The larger the variance of radii is, the slower the computations will run. The reason is two-fold: The smallest particle dictates the time step length, where smaller particles cause shorter time steps. At the same time, the largest particle determines the sorting cell size, where larger particles cause larger cells. Larger cells are likely to contain more particles, causing more contact checks.

Parameters

- **psd** (*str*) – The particle side distribution. One possible value is `logn`, which is a log-normal probability distribution, suitable for approximating well-sorted, coarse sediments. The other possible value is `uni`, which is a uniform distribution from `radius_mean-radius_variance` to `radius_mean+radius_variance`.
- **radius_mean** (*float*) – The mean radius [m] (default = 440e-6 m)
- **radius_variance** (*float*) – The variance in the probability distribution [m].

See also: `generateBimodalRadii()`.

initFluid(*mu=0.00089*)

Initialize the fluid arrays and the fluid viscosity. The default value of `mu` equals the dynamic viscosity of water at 25 degrees Celcius. The value for water at 0 degrees Celcius is 17.87e-4 kg/(m*s).

Parameters **mu** (*float*) – The fluid dynamic viscosity [kg/(m*s)]

initGrid()

Initialize grid suitable for the particle positions set previously. The margin parameter adjusts the distance (in no. of max. radii) from the particle boundaries. *Important*: The particle radii have to be set beforehand.

initGridAndWorldsize(*margin=2.0*)

Initialize grid suitable for the particle positions set previously. The margin parameter adjusts the distance (in no. of max. radii) from the particle boundaries. If the upper wall is dynamic, it is placed at the top boundary of the world.

Parameters **margin** (*float*) – Distance to world boundary in no. of max. particle radii

initGridPos(*gridnum=array([12*, *12*, *36])*)

Initialize particle positions in loose, cubic configuration. `gridnum` is the number of cells in the x, y and z directions. *Important*: The particle radii and the boundary conditions (periodic or not) for the x and y boundaries have to be set beforehand.

Parameters **gridnum** (*numpy.array*) – The number of particles in x, y and z directions

initRandomGridPos(*gridnum=array([12*, *12*, *32])*)

Initialize particle positions in loose, cubic configuration with some variance. `gridnum` is the number of cells in the x, y and z directions. *Important*: The particle radii and the boundary conditions (periodic or not) for the x and y boundaries have to be set beforehand. The world size and grid height (in the z direction) is readjusted to fit the particle positions.

Parameters **gridnum** (*numpy.array*) – The number of particles in x, y and z directions

initRandomPos(*gridnum=array([12*, *12*, *36])*)

Initialize particle positions in completely random configuration. Radii *must* be set beforehand. If the x and y boundaries are set as periodic, the particle centers will be placed all the way to the edge. On regular, non-periodic boundaries, the particles are restrained at the edges to make space for their radii within the bounding box.

Parameters **gridnum** (*numpy.array*) – The number of sorting cells in each spatial direction (default = [12, 12, 36])

initTemporal(*total*, *current=0.0*, *file_dt=0.05*, *step_count=0*)
Set temporal parameters for the simulation. *Important*: Particle radii, physical parameters, and the optional fluid grid need to be set prior to these. The automatically selected value of the computational time step for the DEM is checked for stability in the CFD solution if fluid simulation is included.

Parameters

- **total** (*float*) – The time at which to end the simulation [s]
- **current** – The current time [s] (default = 0.0 s)
- **file_dt** – The interval between output files [s] (default = 0.05 s)

Step_count The number of the first output file (default = 0)

periodicBoundariesX()
Set the x boundary conditions to be periodic.

periodicBoundariesXY()
Set the x and y boundary conditions to be periodic.

plotConvergence(*graphics_format='png'*)
Plot the convergence evolution in the CFD solver. The plot is saved in the output folder with the file name '<simulation id>-conv.<graphics_format>'.

Parameters **graphics_format** (*str*) – Save the plot in this format

plotFluidDiffAdvPresZ(*graphics_format='png'*)
Compare contributions to the velocity from diffusion and advection, assuming the flow is 1D along the z-axis, phi = 1, and dphi = 0. This solution is analog to the predicted velocity and not constrained by the conservation of mass. The plot is saved in the output folder with the name format '<simulation id>-diff_adv-t=<current time>s-mu=<dynamic viscosity>Pa-s.<graphics_format>'.

Parameters **graphics_format** (*str*) – Save the plot in this format

plotFluidPressuresY(*y=-1*, *graphics_format='png'*)
Plot fluid pressures in a plane normal to the second axis. The plot is saved in the current folder with the format 'p_f-<simulation id>-y<y value>.<graphics_format>'.

Parameters

- **y** (*int*) – Plot pressures in fluid cells with these y axis values. If this value is -1, the center y position is used.
- **graphics_format** (*str*) – Save the plot in this format

See also: `writeFluidVTK()` and `plotFluidPressuresZ()`

plotFluidPressuresZ(*z=-1*, *graphics_format='png'*)
Plot fluid pressures in a plane normal to the third axis. The plot is saved in the current folder with the format 'p_f-<simulation id>-z<z value>.<graphics_format>'.

Parameters

- **z** (*int*) – Plot pressures in fluid cells with these z axis values. If this value is -1, the center z position is used.
- **graphics_format** (*str*) – Save the plot in this format

See also: `writeFluidVTK()` and `plotFluidPressuresY()`

plotFluidVelocitiesY(*y=-1*, *graphics_format='png'*)
Plot fluid velocities in a plane normal to the second axis. The plot is saved in the current folder with the format 'v_f-<simulation id>-z<z value>.<graphics_format>'.

Parameters

- **y** (*int*) – Plot velocities in fluid cells with these y axis values. If this value is -1, the center y position is used.
- **graphics_format** (*str*) – Save the plot in this format

See also: `writeFluidVTK()` and `plotFluidVelocitiesZ()`

plotFluidVelocitiesZ(*z=-1*, *graphics_format='png'*)
Plot fluid velocities in a plane normal to the third axis. The plot is saved in the current folder with the format 'v_f-<simulation id>-z<z value>.<graphics_format>'.

Parameters

- **z** (*int*) – Plot velocities in fluid cells with these z axis values. If this value is -1, the center z position is used.
- **graphics_format** (*str*) – Save the plot in this format

See also: `writeFluidVTK()` and `plotFluidVelocitiesY()`

plotPrescribedFluidPressures(*graphics_format='png'*)
Plot the prescribed fluid pressures through time that may be modulated through the class parameters p_mod_A, p_mod_f, and p_mod_phi. The plot is saved in the output folder with the file name '<simulation id>-pres.<graphics_format>'.

porosities(*graphics_format='pdf'*, *zslices=16*)
Plot the averaged porosities with depth. The plot is saved in the format '<simulation id>-porosity.<graphics_format>'.

Parameters

- **graphics_format** (*str*) – Save the plot in this format
- **zslices** (*int*) – The number of points along the vertical axis to sample the porosity in

porosity(*slices=10*, *verbose=False*)
Calculates the porosity as a function of depth, by averaging values in horizontal slabs. Returns porosity values and their corresponding depth. The values are calculated using the external `porosity` program.

Parameters

- **slices** (*int*) – The number of vertical slabs to find porosities in.
- **verbose** (*bool*) – Show the file name of the temporary file written to disk

Returns A 2d array of depths and their averaged porosities

Return type numpy.array

randomBondPairs(*ratio=0.3*, *spacing=-0.1*)
Bond an amount of particles in two-particle clusters. The particles should be initialized beforehand. Note: The actual number of bonds is likely to be somewhat smaller than specified, due to the random selection algorithm.

Parameters

- **ratio** (*float*) – The amount of particles to bond, values in]0.0;1.0]
- **spacing** (*float*) – The distance relative to the sum of radii between bonded particles, neg. values denote an overlap. Values in]0.0,inf[.

readbin (*targetbin*, *verbose=True*, *bonds=True*, *devsmod=True*, *esysparticle=False*)
Reads a target `sphere` binary file.

See also `writebin()`, `readfirst()`, `readlast()`, `readsecond()`, and `readstep()`.

Parameters

- **targetbin** (*str*) – The path to the binary `sphere` file
- **verbose** (*bool*) – Show diagnostic information (default = True)
- **bonds** (*bool*) – The input file contains bond information (default = True). This parameter should be true for all recent `sphere` versions.
- **devsmod** (*bool*) – The input file contains information about modulating stresses at the top wall (default = True). This parameter should be true for all recent `sphere` versions.
- **esysparticle** (*bool*) – Stop reading the file after reading the kinematics, which is useful for reading output files from other DEM programs. (default = False)

readfirst (*verbose=True*)
Read the first output file from the `../output/` folder, corresponding to the object simulation id (`self.sid`).

Parameters **verbose** (*bool*) – Display diagnostic information (default = True)

See also `readbin()`, `readlast()`, `readsecond()`, and `readstep()`.

readlast (*verbose=True*)
Read the last output file from the `../output/` folder, corresponding to the object simulation id (`self.sid`).

Parameters **verbose** (*bool*) – Display diagnostic information (default = True)

See also `readbin()`, `readfirst()`, `readsecond()`, and `readstep()`.

readsecond (*verbose=True*)
Read the second output file from the `../output/` folder, corresponding to the object simulation id (`self.sid`).

Parameters **verbose** (*bool*) – Display diagnostic information (default = True)

See also `readbin()`, `readfirst()`, `readlast()`, and `readstep()`.

readstep (*step*, *verbose=True*)
Read a output file from the `../output/` folder, corresponding to the object simulation id (`self.sid`).

Parameters

- **step** (*int*) – The output file number to read, starting from 0.
- **verbose** (*bool*) – Display diagnostic information (default = True)

See also `readbin()`, `readfirst()`, `readlast()`, and `readsecond()`.

render (*method='pres'*, *max_val=1000.0*, *lower_cutoff=0.0*, *graphics_format='png'*, *verbose=True*)
Using the built-in ray tracer, render all output files that belong to the simulation, determined by the simulation id (`sid`).

Parameters

- **method** (*str*) – The color visualization method to use for the particles. Possible values are: 'normal': color all particles with the same color, 'pres': color by pressure, 'vel': color by translational velocity, 'angvel': color by rotational velocity, 'xdisp': color by total displacement along the x-axis, 'angpos': color by angular position.

- **max_val** (*float*) – The maximum value of the color bar
- **lower_cutoff** (*float*) – Do not render particles with a value below this value, of the field selected by `method`
- **graphics_format** (*str*) – Convert the PPM images generated by the ray tracer to this image format using Imagemagick
- **verbose** (*bool*) – Show verbose information during ray tracing

run (*verbose=True*, *hideinputfile=False*, *dry=False*, *valgrind=False*, *cudamemcheck=False*)
Start `sphere` calculations on the `sim` object

Parameters

- **verbose** (*bool*) – Show `sphere` output
- **hideinputfile** (*bool*) – Hide the file name of the `sphere` input file
- **dry** (*bool*) – Perform a dry run. Important parameter values are shown by the `sphere` program, and it exits afterwards.
- **valgrind** (*bool*) – Run the program with `valgrind` in order to check memory leaks in the host code. This causes a significant increase in computational time.
- **cudamemcheck** (*bool*) – Run the program with `cudamemcheck` in order to check for device memory leaks and errors. This causes a significant increase in computational time.

setFluidPressureModulation (*A*, *f*, *phi=0.0*)
Set the parameters for the sine wave modulating the fluid pressures at the top boundary. Note that a cos-wave is obtained with phi=pi/2.

Parameters

- **A** (*float*) – Fluctuation amplitude [Pa]
- **f** (*float*) – Fluctuation frequency [Hz]
- **phi** (*float*) – Fluctuation phase shift (default=0.0)

See also: `disableFluidPressureModulation()`

shear (*shear_strain_rate=1.0*)
Setup shear experiment. The shear strain rate is the shear velocity divided by the initial height per second. The shear movement is along the positive x axis. The function zeroes the tangential wall viscosity (gamma_wt) and the wall friction coefficients (mu_ws, mu_wn).

Parameters **shear_strain_rate** (*float*) – The shear strain rate to use.

shearStrain ()
Calculates and returns the current shear strain (gamma) value of the experiment. The shear strain is found by determining the total x-axis displacement of the upper, fixed particles.

Returns The total shear strain [-]

Return type float

shearVel ()
Calculates and returns the shear velocity (gamma_dot) of the experiment. The shear velocity is the x-axis velocity value of the upper particles.

Returns The shear velocity applied by the upper, fixed particles [m/s]

Return type float

sheardisp(*graphics_format='pdf'*, *zslices=32*)
Plot the particle x-axis displacement against the original vertical particle position. The plot is saved in the current directory with the file name '<simulation id>-sheardisp.<graphics_format>'.

Parameters **graphics_format** (*str*) – Save the plot in this format

status()
Returns the current simulation status by using the simulation id (`sid`) as an identifier.

Returns The number of the last output file written

Return type int

thinsection_x1x3(*x2='center'*, *graphics_format='png'*, *cbmax=None*, *arrowscale=0.01*, *velarrowscale=1.0*, *slipscale=1.0*, *verbose=False*)
Produce a 2D image of particles on a x1,x3 plane, intersecting the second axis at x2. Output is saved as '<sid>-ts-x1x3.txt' in the current folder.

An upper limit to the pressure color bar range can be set by the cbmax parameter.

The data can be plotted in gnuplot with: gnuplot> set size ratio -1 gnuplot> set palette defined (0 "blue", 0.5 "gray", 1 "red") gnuplot> plot '<sid>-ts-x1x3.txt' with circles palette fs transparent solid 0.4 noborder

This function also saves a plot of the inter-particle slip angles.

Parameters

- **x2** (*foat*) – The position along the second axis of the intersecting plane
- **graphics_format** (*str*) – Save the slip angle plot in this format
- **cbmax** (*float*) – The maximal value of the pressure color bar range
- **arrowscale** (*float*) – Scale the rotational arrows by this value
- **velarrowscale** (*float*) – Scale the translational arrows by this value
- **slipscale** (*float*) – Scale the slip arrows by this value
- **verbose** (*bool*) – Show function output during calculations

torqueScript(*email='adc@geo.au.dk'*, *email_alerts='ae'*, *walltime='24:00:00'*, *queue='qfermi'*, *cudapath='/com/cuda/4.0.17/cuda'*, *spheredir='/home/adc/code/sphere'*, *use_workdir=False*, *workdir='/scratch'*)
Creates a job script for the Torque queue manager for the simulation object.

Parameters

- **email** (*str*) – The e-mail address that Torque messages should be sent to
- **email_alerts** (*str*) – The type of Torque messages to send to the e-mail address. The character 'b' causes a mail to be sent when the execution begins. The character 'e' causes a mail to be sent when the execution ends normally. The character 'a' causes a mail to be sent if the execution ends abnormally. The characters can be written in any order.
- **walltime** (*str*) – The maximal allowed time for the job, in the format 'HH:MM:SS'.
- **queue** (*str*) – The Torque queue to schedule the job for
- **cudapath** (*str*) – The path of the CUDA library on the cluster compute nodes
- **spheredir** (*str*) – The path to the root directory of sphere on the cluster
- **use_workdir** (*bool*) – Use a different working directory than the sphere folder
- **workdir** (*str*) – The working directory during the calculations, if *use_workdir=True*

totalMomentum()
Returns the sum of particle momentums.

Returns The sum of particle momentums (m*v) [N*s]

Return type float

triaxial(*wvel=-0.001*, *normal_stress=10000.0*)
Setup triaxial experiment. The upper wall is moved at a fixed velocity in m/s, default values is -0.001 m/s (i.e. downwards). The side walls are exerting a defined normal stress.

Parameters

- **wvel** (*float*) – Upper wall velocity. Negative values mean that the wall moves downwards.
- **normal_stress** (*float*) – The normal stress to apply from the upper wall.

uniaxialStrainRate(*wvel=-0.001*)
Setup consolidation experiment. Specify the upper wall velocity in m/s, default value is -0.001 m/s (i.e. downwards).

Parameters **wvel** (*float*) – Upper wall velocity. Negative values mean that the wall moves downwards.

video(*out_folder='./'*, *video_format='mp4'*, *graphics_folder='../img_out/'*, *graphics_format='png'*, *fps=25*, *qscale=1*, *bitrate=1800*, *verbose=False*)
Uses ffmpeg to combine images to animation. All images should be rendered beforehand using func:*render()*.

Parameters

- **out_folder** (*str*) – The output folder for the video file
- **video_format** (*str*) – The format of the output video
- **graphics_folder** (*str*) – The folder containing the rendered images
- **graphics_format** (*str*) – The format of the rendered images
- **fps** (*int*) – The number of frames per second to use in the video
- **qscale** (*float*) – The output video quality, in]0;1]
- **bitrate** (*int*) – The bitrate to use in the output video
- **verbose** (*bool*) – Show ffmpeg output

voidRatio()
Calculates the current void ratio

Returns The void ratio, in [0:1]

Return type float

writeFluidVTK(*folder='../output/'*, *verbose=True*)
Writes a VTK file for the fluid grid to the `../output/` folder by default. The file name will be in the format `fluid-<self.sid>.vti`. The vti files can be used for visualizing the fluid in ParaView.

The fluid grid is visualized by opening the vti files, and pressing "Apply" to import all fluid field properties. To visualize the scalar fields, such as the pressure, the porosity, the porosity change or the velocity magnitude, choose "Surface" or "Surface With Edges" as the "Representation". Choose the desired property as the "Coloring" field. It may be desirable to show the color bar by pressing the "Show" button, and "Rescale" to fit the color range limits to the current file. The coordinate system can be displayed by checking the "Show Axis" field. All adjustments by default require the "Apply" button to be pressed before regenerating the view.

The fluid vector fields (e.g. the fluid velocity) can be visualizing by e.g. arrows. To do this, select the fluid data in the "Pipeline Browser". Press "Glyph" from the "Common" toolbar, or go to the "Filters" mennu, and press "Glyph" from the "Common" list. Make sure that "Arrow" is selected as the "Glyph type", and "Velocity" as the "Vectors" value. Adjust the "Maximum Number of Points" to be at least as big as the number of fluid cells in the grid. Press "Apply" to visualize the arrows.

If several data files are generated for the same simulation (e.g. using the `writeVTKall()` function), it is able to step the visualization through time by using the ParaView controls.

Parameters

- **folder** (*str*) – The folder where to place the output binary file (default (default = '../output/')
- **verbose** (*bool*) – Show diagnostic information (default = True)

writeVTK(*folder='../output/'*, *verbose=True*)

Writes a VTK file with particle information to the `../output/` folder by default. The file name will be in the format `<self.sid>.vtu`. The vtu files can be used to visualize the particles in ParaView.

After opening the vtu files, the particle fields will show up in the "Properties" list. Press "Apply" to import all fields into the ParaView session. The particles are visualized by selecting the imported data in the "Pipeline Browser". Afterwards, click the "Glyph" button in the "Common" toolbar, or go to the "Filters" menu, and press "Glyph" from the "Common" list. Choose "Sphere" as the "Glyph Type", set "Radius" to 1.0, choose "scalar" as the "Scale Mode". Check the "Edit" checkbox, and set the "Set Scale Factor" to 1.0. The field "Maximum Number of Points" may be increased if the number of particles exceed the default value. Finally press "Apply", and the particles will appear in the main window.

The sphere resolution may be adjusted ("Theta resolution", "Phi resolution") to increase the quality and the computational requirements of the rendering. All adjustments by default require the "Apply" button to be pressed before regenerating the view.

If several vtu files are generated for the same simulation (e.g. using the :func:`writeVTKall()` function), it is able to step the visualization through time by using the ParaView controls.

Parameters

- **folder** (*str*) – The folder where to place the output binary file (default (default = '../output/')
- **verbose** (*bool*) – Show diagnostic information (default = True)

writeVTKall(*verbose=True*)

Writes a VTK file for each simulation output file with particle information and the fluid grid to the `../output/` folder by default. The file name will be in the format `<self.sid>.vtu` and `fluid-<self.sid>.vti`. The vtu files can be used to visualize the particles, and the vti files for visualizing the fluid in ParaView.

After opening the vtu files, the particle fields will show up in the "Properties" list. Press "Apply" to import all fields into the ParaView session. The particles are visualized by selecting the imported data in the "Pipeline Browser". Afterwards, click the "Glyph" button in the "Common" toolbar, or go to the "Filters" menu, and press "Glyph" from the "Common" list. Choose "Sphere" as the "Glyph Type", set "Radius" to 1.0, choose "scalar" as the "Scale Mode". Check the "Edit" checkbox, and set the "Set Scale Factor" to 1.0. The field "Maximum Number of Points" may be increased if the number of particles exceed the default value. Finally press "Apply", and the particles will appear in the main window.

The sphere resolution may be adjusted ("Theta resolution", "Phi resolution") to increase the quality and the computational requirements of the rendering.

The fluid grid is visualized by opening the vti files, and pressing "Apply" to import all fluid field properties. To visualize the scalar fields, such as the pressure, the porosity, the porosity change or the velocity

magnitude, choose "Surface" or "Surface With Edges" as the "Representation". Choose the desired property as the "Coloring" field. It may be desirable to show the color bar by pressing the "Show" button, and "Rescale" to fit the color range limits to the current file. The coordinate system can be displayed by checking the "Show Axis" field. All adjustments by default require the "Apply" button to be pressed before regenerating the view.

The fluid vector fields (e.g. the fluid velocity) can be visualizing by e.g. arrows. To do this, select the fluid data in the "Pipeline Browser". Press "Glyph" from the "Common" toolbar, or go to the "Filters" mennu, and press "Glyph" from the "Common" list. Make sure that "Arrow" is selected as the "Glyph type", and "Velocity" as the "Vectors" value. Adjust the "Maximum Number of Points" to be at least as big as the number of fluid cells in the grid. Press "Apply" to visualize the arrows.

If several data files are generated for the same simulation (e.g. using the `writeVTKall()` function), it is able to step the visualization through time by using the ParaView controls.

Parameters **verbose** (*bool*) – Show diagnostic information (default = True)

writebin(*folder='../input/'*, *verbose=True*)

Writes a `sphere` binary file to the `../input/` folder by default. The file name will be in the format `<self.sid>.bin`.

See also `readbin()`.

Parameters

- **folder** (*str*) – The folder where to place the output binary file
- **verbose** (*bool*) – Show diagnostic information (default = True)

zeroKinematics()

Zero all kinematic parameters of the particles. This function is useful when output from one simulation is reused in another simulation.

sphere.**status**(*project*)

Check the status.dat file for the target project, and return the last output file number.

Parameters **project** (*str*) – The simulation id of the target project

Returns The last output file written in the simulation calculations

Return type int

sphere.**thinsectionVideo**(*project*, *out_folder='./'*, *video_format='mp4'*, *fps=25*, *qscale=1*, *bitrate=1800*, *verbose=False*)

Uses ffmpeg to combine thin section images to an animation. This function will implicity render the thin section images beforehand.

Parameters

- **project** (*str*) – The simulation id of the project to render
- **out_folder** (*str*) – The output folder for the video file
- **video_format** (*str*) – The format of the output video
- **fps** (*int*) – The number of frames per second to use in the video
- **qscale** (*float*) – The output video quality, in]0;1]
- **bitrate** (*int*) – The bitrate to use in the output video
- **verbose** (*bool*) – Show ffmpeg output

sphere.**torqueScriptParallel3**(*obj1*, *obj2*, *obj3*, *email='adc@geo.au.dk'*, *email_alerts='ae'*, *walltime='24:00:00'*, *queue='qfermi'*, *cudapath='/com/cuda/4.0.17/cuda'*, *spheredir='/home/adc/code/sphere'*, *use_workdir=False*, *workdir='/scratch'*)

Create job script for the Torque queue manager for three binaries, executed in parallel, ideally on three GPUs.

Parameters

- **email** (*str*) – The e-mail address that Torque messages should be sent to
- **email_alerts** (*str*) – The type of Torque messages to send to the e-mail address. The character 'b' causes a mail to be sent when the execution begins. The character 'e' causes a mail to be sent when the execution ends normally. The character 'a' causes a mail to be sent if the execution ends abnormally. The characters can be written in any order.
- **walltime** (*str*) – The maximal allowed time for the job, in the format 'HH:MM:SS'.
- **queue** (*str*) – The Torque queue to schedule the job for
- **cudapath** (*str*) – The path of the CUDA library on the cluster compute nodes
- **spheredir** (*str*) – The path to the root directory of sphere on the cluster
- **use_workdir** (*bool*) – Use a different working directory than the sphere folder
- **workdir** (*str*) – The working directory during the calculations, if *use_workdir=True*

Returns The filename of the script

Return type str

See also `torqueScript()`

sphere.**vector_norm**(*ndvector*)

Returns a 1D vector of normalized values. The input array should have one row per particle, and three rows; one per Euclidean axis.

Returns A value of the velocity magnutude per particle

Return type numpy.array

sphere.**video**(*project*, *out_folder='./'*, *video_format='mp4'*, *graphics_folder='../img_out/'*, *graphics_format='png'*, *fps=25*, *qscale=1*, *bitrate=1800*, *verbose=False*)

Uses ffmpeg to combine images to animation. All images should be rendered beforehand using func:*render()*.

Parameters

- **project** (*str*) – The simulation id of the project to render
- **out_folder** (*str*) – The output folder for the video file
- **video_format** (*str*) – The format of the output video
- **graphics_folder** (*str*) – The folder containing the rendered images
- **graphics_format** (*str*) – The format of the rendered images
- **fps** (*int*) – The number of frames per second to use in the video
- **qscale** (*float*) – The output video quality, in]0;1]
- **bitrate** (*int*) – The bitrate to use in the output video
- **verbose** (*bool*) – Show ffmpeg output

sphere.**visualize**(*project*, *method='energy'*, *savefig=True*, *outformat='png'*)

Visualize output from the target project, where the temporal progress is of interest. The output will be saved in the current folder with a name combining the simulation id of the project, and the visualization method.

Parameters

- **project** (*str*) – The simulation id of the project to render
- **method** (*str*) – The type of plot to render. Possible values are 'energy', 'walls', 'triaxial' and 'shear'
- **savefig** (*bool*) – Save the image instead of showing it on screen
- **outformat** – The output format of the plot data. This can be an image format, or in text ('txt').

CHAPTER

TWO

INDICES AND TABLES

- *genindex*
- *search*

PYTHON MODULE INDEX

S

INDEX

T

U

V

W

Z