sphere Documentation

Release 2.15-beta

Anders Damsgaard

Sep 07, 2017

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, https://adamsgaard.dk, mailto:anders.damsgaard@geo.au.dk

ONE

CONTENTS

1.1 Introduction and Installation

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 8.0 or newer

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

```
$ sudo apt-get install build-essential cmake nvidia-cuda-toolkit clang-3.8
```

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 2.0 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 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.

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

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

NOTE: If your system does not have a GCC compiler compatible with the installed CUDA version (e.g. GCC-5 for CUDA 8), you will see errors at the linker stage. In that case, try using `clang-3.8` as the C and C++ compiler instead:

```
$ rm -rf CMakeCache.txt CMakeFiles/
$ export CC=$(which clang-3.8) && export CXX=$(which clang++-3.8) && cmake . && make
```

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
```

Disclaimer: On some systems the Navier-Stokes related tests will fail. If you do encounter these problems, but do not plan on using the Navier Stokes solver for fluid dynamics, carry on.

If successful 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:

```
.----------------------------------.
|                _                 |
|               | |                |
|     ___ _ __  | |__   ___ _ __ ___     |
|    / __| '_ \| '_ \ / _ \ '__/ _ \  |
|    \__ \ |_) | | | |  __/ | |  __/   |
|    |___/ .__/|_| |_|\___|_|  \___|   |
|        | |                       |
|        |_|          Version: 2.15   |
`----------------------------------´
 A discrete-element method particle dynamics simulator.
 Written by Anders Damsgaard, license GPLv3+.
 https://adamsgaard.dk
```

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 (`python/sphere.py`).

- 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:

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

Here, $\boldsymbol{v}$ is the fluid velocity, ρ is the fluid density, $\boldsymbol{\sigma}$ is the Cauchy stress tensor, $\boldsymbol{f}^i$ is the particle-fluid interaction vector and $\boldsymbol{g}$ is the gravitational acceleration. 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:

$$\frac{D(\phi v_x)}{Dt} = \frac{1}{\rho} [\nabla \cdot (\phi \boldsymbol{\sigma})]_x - \frac{1}{\rho} f_x^i + \phi g$$

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

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

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

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

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:

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

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:

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

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

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:

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

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.

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

$$\Downarrow$$

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

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

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

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

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).

$$v_x^* = v_x^t + \Delta v_x$$

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

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):

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

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

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

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

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

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

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

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:

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

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

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:

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

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

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

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:

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

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

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.:

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

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.:

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

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`.

```python
#!/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)
```

```
32
33   # Define the temporal parameters, e.g. the total time (total) and the file
34   # output interval (file_dt), both in seconds
35   SB.initTemporal(total = 2.0, file_dt = 0.1)
36
37   # Using a 'dry' run, the sphere main program will display important parameters.
38   # sphere will end after displaying these values.
39   SB.run(dry = True)
40
41
42   ### RUNNING THE SIMULATION
43
44   # Start the simulation on the GPU from the sphere program
45   SB.run()
46
47
48   ### ANALYSIS OF SIMULATION RESULTS
49
50   # Plot the system energy through time, image saved as collision-energy.png
51   SB.visualize(method = 'energy')
52
53   # Render the particles using the built-in raytracer
54   SB.render()
55
56   # Alternative visualization using ParaView. See the documentation of
57   # ``sim.writeVTKall()`` for more information about displaying the
58   # particles in ParaView.
59   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*, *cfd_solver=0*)
  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)

  - **cfd_solver** (`int`) – Fluid solver to use if fluid == True. 0: Navier-Stokes (default), 1: Darcy.

ReynoldsNumber()
  Estimate the per-cell Reynolds number by: Re = rho * ||v_f|| * dx/mu. This value is returned and also stored in *self.Re*.

  Returns Reynolds number

  Return type Numpy array with dimensions like the fluid grid

acceleration(*idx=-1*)
  Returns the acceleration of one or more particles, selected by their index. If the index is equal to -1 (default value), all accelerations are returned.

  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

adaptiveGrid()
:   Set the height of the fluid grid to automatically readjust to the height of the granular assemblage, as dictated by the position of the top wall. This will readjust *self.L[2]* during the simulation to equal the position of the top wall *self.w_x[0]*.

    See also *staticGrid()*

addParticle(*x*, *radius*, *xyzsum=array([0., 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.])*, *color=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*) – 0: Do not fix particle velocity (default), 1: Fix horizontal linear velocity, -1: Fix horizontal and vertical linear velocity
    - **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*, *wall_mass_factor=1.0*)
:   Adjust grid and dynamic wall to max. particle position. The wall thickness will by standard equal the maximum particle diameter. The density equals the particle density, and the wall size is equal to the width and depth of the simulation domain (*self.L[0]* and *self.L[1]*). The total wall mass can be linearly scaled by the *wall_mass_factor* parameter.

    > **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.

    :param

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(*trim=True*)

> Calculates the bulk porosity of the particle assemblage.
>
> > Parameters **trim** (*bool*) – Trim the total volume to the smallest axis-parallel cube containing all particles.
> >
> > Returns The bulk porosity, in [0:1]
> >
> > Return type float

cellSize()

> Calculate the particle sorting (and fluid) cell dimensions. These values are stored in *self.dx* and are NOT returned.

checkerboardColors(*nx=6*, *ny=6*, *nz=6*)

> Assign checkerboard color values to the particles in an orthogonal grid.
>
> > Parameters
> >
> > - **nx** (*int*) – Number of color values along the x axis
> > - **ny** (*int*) – Number of color values along the y ayis
> > - **nz** (*int*) – Number of color values along the z azis

cleanup()

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

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)

contactParticleArea(*i*, *j*)

> Finds the average area of an two particles in an inter-particle contact.
>
> > Parameters
> >
> > - **i** (*int or array of ints*) – Index of first particle
> > - **j** (*int or array of ints*) – Index of second particle
> > - **d** (*float or array of floats*) – Overlap distance
> >
> > Returns Contact area [m*m]

> **Return type** float or array of floats

contactSurfaceArea(*i*, *j*, *overlap*)

Finds the contact surface area of an inter-particle contact.

> **Parameters**
>
> - **i** (*int or array of ints*) – Index of first particle
>
> - **j** (*int or array of ints*) – Index of second particle
>
> - **d** (*float or array of floats*) – Overlap distance
>
> **Returns** Contact area [m*m]
>
> **Return type** float or array of floats

convergence()

Read the convergence evolution in the CFD solver. The values are stored in *self.conv* with iteration number in the first column and iteration count in the second column.

> See also: *plotConvergence()*

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(*type='defined'*)

Calculates the current magnitude of the defined or effective top wall normal stress.

> **Parameters** **type** (*str*) – Find the 'defined' (default) or 'effective' normal stress
>
> **Returns** The current top wall normal stress in Pascal
>
> **Return type** float

currentTime(*value=-1*)

Get or set the current time. If called without arguments the current time is returned. If a new time is passed in the 'value' argument, the time is written to the object.

> **Parameters** **value** (*float*) – The new current time
>
> **Returns** The current time
>
> **Return type** float

defaultParams(*mu_s=0.5*, *mu_d=0.5*, *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], dx=-1*)

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].
>
> - **dx** (*float*) – The cell width in any direction. If the default value is used (-1), the cell width is calculated to fit the largest particle.

deleteAllParticles ()

Deletes all particles in the simulation object.

deleteParticle (*i*)

Delete particle(s) with index i.

> **Parameters i** (*int, list or numpy.array*) – One or more particle indexes to delete

disableFluidPressureModulation ()

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

See also: *setFluidPressureModulation()*

disableTopWallNormalStressModulation ()

Set the parameters for the sine wave modulating the normal stress at the top dynamic wall to zero.

See also: *setTopWallNormalStressModulation()*

dry ()

Set the simulation to be dry (no fluids).

See also *wet()*

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

findAllAverageParticlePairAreas()
> Finds the average area of an inter-particle contact. This function requires a prior call to *findOverlaps()* as it reads from the `self.pairs` and `self.overlaps` arrays.
>
> > **Returns** Array of contact surface areas
> >
> > **Return type** array of floats

findAllContactSurfaceAreas()
> Finds the contact surface area of an inter-particle contact. This function requires a prior call to *findOverlaps()* as it reads from the `self.pairs` and `self.overlaps` arrays.
>
> > **Returns** Array of contact surface areas
> >
> > **Return type** array of floats

findContactStresses(*area='average'*)
> Finds all particle-particle uniaxial normal stresses (by first calling *findNormalForces()*) and calculating the stress magnitudes by dividing the normal force magnitude with the average particle area ('average') or by the contact surface area ('contact').
>
> The result is saved in `self.sigma_contacts`.
>
> > **Parameters** **area** (*str*) – Area to use: 'average' (default) or 'contact'
>
> See also: *findNormalForces()* and *findOverlaps()*

findCoordinationNumber()
> Finds the coordination number (the average number of contacts per particle). Requires a previous call to *findOverlaps()*. Values are stored in `self.coordinationnumber`.

findHydraulicConductivities()
> Calculates the hydrological conductivities from the Kozeny-Carman relationship. These values are only relevant when the Darcy solver is used (*self.cfd_solver = 1*). The permeability pre-factor *self.k_c* and the assemblage porosities must be set beforehand. The former values are set if a file from the *output/* folder is read using *self.readbin*.

findLoadedContacts(*threshold*)
> Finds the indices of contact pairs where the contact stress magnitude exceeds or is equal to a specified threshold value. This function calls *findContactStresses()*.
>
> > **Parameters** **threshold** (*float*) – Threshold contact stress [Pa]
> >
> > **Returns** Array of contact indices
> >
> > **Return type** array of ints

findMeanCoordinationNumber()
> Returns the coordination number (the average number of contacts per particle). Requires a previous call to *findOverlaps()*
>
> > **Returns** The mean particle coordination number
> >
> > **Return type** float

findNormalForces()
> Finds all particle-particle overlaps (by first calling *findOverlaps()*) and calculating the normal magnitude by multiplying the overlaps with the elastic stiffness `self.k_n`.
>
> The result is saved in `self.f_n_magn`.

See also: *findOverlaps()* and *findContactStresses()*

findOverlaps()
　　Find all particle-particle overlaps by a n^2 contact search, which is done in C++. The particle pair indexes and the distance of the overlaps is saved in the object itself as the `.pairs` and `.overlaps` members.

　　See also: *findNormalForces()*

findPermeabilities()
　　Calculates the hydrological permeabilities from the Kozeny-Carman relationship. These values are only relevant when the Darcy solver is used (*self.cfd_solver = 1*). The permeability pre-factor *self.k_c* and the assemblage porosities must be set beforehand. The former values are set if a file from the *output/* folder is read using *self.readbin*.

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

frictionalEnergy(*idx*)
　　Returns the frictional dissipated energy for a particle.

　　　Parameters idx (*int*) – Particle index

　　　Returns The frictional energy lost of the particle [J]

　　　Return type float

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'*, *mean=0.00044*, *variance=8.8e-09*, *histogram=False*)
> 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 `mean -variance` to `mean + variance`.
> >
> > - **mean** (`float`) – The mean radius [m] (default = 440e-6 m)
> >
> > - **variance** (`float`) – The variance in the probability distribution [m].
>
> See also: *generateBimodalRadii()*.

hydraulicConductivity(*phi=0.35*)
> Determine the hydraulic conductivity (K) [m/s] from the permeability prefactor and a chosen porosity. This value is stored in *self.K_c*. This function only works for the Darcy solver (*self.cfd_solver == 1*)
>
> > **Parameters** **phi** (`float`) – The porosity to use in the Kozeny-Carman relationship
> >
> > **Returns** The hydraulic conductivity [m/s]
> >
> > **Return type** float

hydraulicDiffusivity()
> Determine the hydraulic diffusivity (D) [m*m/s]. The result is stored in *self.D*. This function only works for the Darcy solver (*self.cfd_solver[0] == 1*)

hydraulicPermeability(*phi_min=0.3*, *phi_max=0.99*)
> Determine the hydraulic permeability (k) [m*m] from the Kozeny-Carman relationship, using the permeability prefactor (*self.k_c*), and the range of valid porosities set in *src/darcy.cuh*, by default in the range 0.1 to 0.9.
>
> This function is only valid for the Darcy solver (*self.cfd_solver == 1*).

id(*sid=''*)
> Returns or sets the simulation id/name, which is used to identify simulation files in the output folders.
>
> > **Parameters** **sid** (`str`) – The desired simulation id. If left blank the current simulation id will be returned.
> >
> > **Returns** The current simulation id if no new value is set.
> >
> > **Return type** str

idAppend(*string*)
> Append a string to the simulation id/name, which is used to identify simulation files in the output folders.
>
> > **Parameters** **string** (`str`) – The string to append to the simulation id (*self.sid*).

inertiaParameterPlanarShear()
> Returns the value of the inertia parameter I during planar shear proposed by GDR-MiDi 2004.
>
> > **Returns** The value of I
> >
> > **Return type** float
>
> See also: *shearStrainRate()* and *shearVel()*

initFluid(*mu=0.00089*, *rho=1000.0*, *p=0.0*, *hydrostatic=False*, *cfd_solver=0*)

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)]
>
> - **rho** (`float`) – The fluid density [kg/(m^3)]
>
> - **p** – The hydraulic pressure to initialize the cells to. If the parameter *hydrostatic* is set to *True*, this value will apply to the fluid cells at the top
>
> - **hydrostatic** (`bool`) – Initialize the fluid pressures to the hydrostatic pressure distribution. A pressure gradient with depth is only created if a gravitational acceleration along z previously has been specified
>
> - **cfd_solver** (`int`) – Solver to use for the computational fluid dynamics. Accepted values: 0 (Navier Stokes, default) and 1 (Darcy).

initGrid(*dx=-1*)

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 if the cell width isn't specified by *dx*.

> **Parameters dx** (`float`) – The cell width in any direction. If the default value is used (-1), the cell width is calculated to fit the largest particle.

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])*, *padding=2.1*)

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
>
> - **padding** (`float`) – Increase distance between particles in x, y and z directions with this multiplier. Large values create more random packings.

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])

- **dx** (*float*) – The cell width in any direction. If the default value is used (-1), the cell width is calculated to fit the largest particle.

initTemporal(*total*, *current=0.0*, *file_dt=0.05*, *step_count=0*, *dt=-1*, *epsilon=0.01*)

Set temporal parameters for the simulation. *Important*: Particle radii, physical parameters, and the optional fluid grid need to be set prior to these if the computational time step (dt) isn't set explicitly. If the parameter *dt* is the default value (-1), the function will estimate the best time step length. The 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)

- **dt** – The computational time step length [s]

- **epsilon** (*float*) – Time step multiplier (default = 0.01)

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

kineticEnergy(*idx*)

Returns the (linear) kinetic energy for a particle.

Parameters **idx** (*int*) – Particle index

Returns The kinetic energy of the particle [J]

Return type float

largestFluidTimeStep(*safety=0.5*, *v_max=-1.0*)

Finds and returns the largest time step in the fluid phase by von Neumann and Courant-Friedrichs-Lewy analysis given the current velocities. This ensures stability in the diffusive and advective parts of the momentum equation.

The value of the time step decreases with increasing fluid viscosity (*self.mu*), and increases with fluid cell size (*self.L/self.num*)

and fluid velocities (*self.v_f*)

Parameters

- **safety** (*float*) – Safety factor which is multiplied to the largest time step.

- **v_max** (*float*) – The largest anticipated absolute fluid velocity [m/s]

Returns The largest timestep stable for the current fluid state.

Return type float

largestMass()

Returns the mass of the heaviest particle.

Parameters **idx** (*int*) – Particle index

Returns The mass of the particle [kg]

Return type float

mass(*idx*)

Returns the mass of a particle.

> **Parameters idx** (*int*) – Particle index
>
> **Returns** The mass of the particle [kg]
>
> **Return type** float

momentOfInertia(*idx*)

Returns the moment of inertia of a particle.

> **Parameters idx** (*int*) – Particle index
>
> **Returns** The moment of inertia [kg*m^2]
>
> **Return type** float

momentum(*idx*)

Returns the momentum (m*v) of a particle.

> **Parameters idx** (*int*) – The particle index
>
> **Returns** The particle momentum [N*s]
>
> **Return type** numpy.array

normalBoundariesXY()

Set the x and y boundary conditions to be static walls.

> See also *periodicBoundariesXY()* and *periodicBoundariesX()*

periodicBoundariesX()

Set the x boundary conditions to be periodic.

> See also *normalBoundariesXY()* and *periodicBoundariesXY()*

periodicBoundariesXY()

Set the x and y boundary conditions to be periodic.

> See also *normalBoundariesXY()* and *periodicBoundariesX()*

plotContacts(*graphics_format='png', figsize=[4, 4], title=None, lower_limit=0.0, upper_limit=1.0, alpha=1.0, return_data=False, outfolder='.', f_min=None, f_max=None, histogram=True, forcechains=True*)

Plot current contact orientations on polar plot

> **Parameters**
>
> - **lower_limit** (*float*) – Do not visualize force chains below this relative contact force magnitude, in]0;1[
>
> - **upper_limit** (*float*) – Visualize force chains above this relative contact force magnitude but cap color bar range, in]0;1[
>
> - **graphics_format** (*str*) – Save the plot in this format

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

> See also: *convergence()*

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()*

plotLoadCurve (*graphics_format='png'*)
Plot the load curve (log time vs. upper wall movement). The plot is saved in the current folder with the file name '<simulation id>-loadcurve.<graphics_format>'. The consolidation coefficient calculations are done on the base of Bowles 1992, p. 129–139, using the "Casagrande" method. It is assumed that the consolidation has stopped at the end of the simulation (i.e. flat curve).

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

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>'.

plotSinFunction (*baseval*, *A*, *f*, *phi=0.0*, *xlabel='t [s]'*, *ylabel='y'*, *plotstyle='.'*, *outformat='png'*)

Plot the values of a sinusoidal modulated base value. Saves the output as a plot in the current folder. The time values will range from *self.time_current* to *self.time_total*.

> **Parameters**
>
> - **baseval** (*float*) – The center value which the sinusoidal fluctuations are modulating
>
> - **A** (*float*) – The fluctuation amplitude
>
> - **phi** (*float*) – The phase shift [s]
>
> - **xlabel** (*str*) – The label for the x axis
>
> - **ylabel** (*str*) – The label for the y axis
>
> - **plotstyle** (*str*) – Matplotlib-string for specifying plotting style
>
> - **outformat** (*str*) – File format of the output plot

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[.

readTime (*time*, *verbose=True*)

Read the output file most closely corresponding to the time given as an argument.

> **Parameters** **time** (*float*) – The desired current time [s]
>
> See also *readbin()*, *readfirst()*, *readsecond()*, and *readstep()*.

---

readbin (*targetbin*, *verbose=True*, *bonds=True*, *sigma0mod=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.

- **sigma0mod** (`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

rotationalEnergy(*idx*)

Returns the rotational energy for a particle.

> **Parameters idx** (*int*) – Particle index
>
> **Returns** The rotational kinetic energy of the particle [J]
>
> **Return type** float

run(*verbose=True*, *hideinputfile=False*, *dry=False*, *valgrind=False*, *cudamemcheck=False*, *device=-1*)

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.
>
> - **device** (*int*) – Specify the GPU device to execute the program on. If not specified, sphere will use the device with the most CUDA cores. To see a list of devices, run nvidia-smi in the system shell.

scaleSize(*factor*)

Scale the positions, linear velocities, forces, torques and radii of all particles and mobile walls.

> **Parameters factor** (*float*) – Spatial scaling factor]0;inf[

setBeta(*beta*)

Beta is a fluid solver parameter, used in velocity prediction and pressure iteration 1.0: Use old pressures for fluid velocity prediction (see Langtangen et al. 2002) 0.0: Do not use old pressures for fluid velocity prediction (Chorin's original projection method, see Chorin (1968) and "Projection method (fluid dynamics)" page on Wikipedia. The best results precision and performance-wise are obtained by using a beta of 0 and a low tolerance criteria value.

The default and recommended value is 0.0.

Other solver parameter setting functions: *setGamma()*, *setTheta()*, *setTolerance()*, *setDEMstepsPerCFDstep()* and *setMaxIterations()*

setDEMstepsPerCFDstep(*ndem*)

A fluid solver parameter, the value of the maxiter parameter denotes the number of DEM time steps to be performed per CFD time step.

The default value is 1.

> **Parameters ndem** (*int*) – The DEM/CFD time step ratio

Other solver parameter setting functions: *setGamma()*, *setTheta()*, *setBeta()*, *setTolerance()* and *setMaxIterations()*.

setDampingNormal(*gamma*, *over_damping=False*)
Set the dampening coefficient (gamma) in the normal direction of the particle-particle contact model. The function will print the fraction between the chosen damping and the critical damping value.

> **Parameters**
>
> • **gamma** (`float`) – The viscous damping constant [N/(m/s)]
>
> • **over_damping** (`boolean`) – Accept overdampening

See also: `setDampingTangential(gamma)()`

setDampingTangential(*gamma*, *over_damping=False*)
Set the dampening coefficient (gamma) in the tangential direction of the particle-particle contact model. The function will print the fraction between the chosen damping and the critical damping value.

> **Parameters**
>
> • **gamma** (`float`) – The viscous damping constant [N/(m/s)]
>
> • **over_damping** (`boolean`) – Accept overdampening

See also: `setDampingNormal(gamma)()`

setDynamicFriction(*mu_d*)
Set the dynamic friction coefficient for particle-particle interactions (*self.mu_d*). This value describes the resistance to a shearing motion while it is happening (contact tangential velocity larger than 0). Strain softening can be introduced by having a smaller dynamic frictional coefficient than the static friction coefficient. Usually this value is identical to the static friction coefficient.

> **Parameters mu_d** (`float`) – Value of the dynamic friction coefficient, in [0;inf[. Usually between 0 and 1.

See also: `setStaticFriction(mu_s)()`

setFluidBottomFixedFlux(*specific_flux*)
Define a constant fluid flux normal to the boundary.

The default behavior for the boundary is fixed value (Dirichlet), see *setFluidBottomFixedPressure()*.

> **Parameters specific_flux** – Specific flux values across boundary (positive values upwards), [m/s]

setFluidBottomFixedPressure()
Set the lower boundary of the fluid domain to follow the fixed pressure value (Dirichlet) boundary condition.

This is the default behavior for the boundary. See also *setFluidBottomNoFlow()*

setFluidBottomNoFlow()
Set the lower boundary of the fluid domain to follow the no-flow (Neumann) boundary condition with free slip parallel to the boundary.

The default behavior for the boundary is fixed value (Dirichlet), see *setFluidBottomFixedPressure()*.

setFluidBottomNoFlowNoSlip()
Set the lower boundary of the fluid domain to follow the no-flow (Neumann) boundary condition with no slip parallel to the boundary.

The default behavior for the boundary is fixed value (Dirichlet), see *setFluidBottomFixedPressure()*.

setFluidCompressibility(*beta_f*)

Set the fluid adiabatic compressibility [1/Pa]. This value is equal to *1/K* where *K* is the bulk modulus [Pa]. The value for water is 5.1e-10 for water at 0 degrees Celcius. This parameter is used for the Darcy solver exclusively.

> **Parameters beta_f** (*float*) – The fluid compressibility [1/Pa]

See also: *setFluidDensity()* and *setFluidViscosity()*

setFluidDensity(*rho_f*)

Set the fluid density [kg/(m*m*m)]. The value for water is 1000. This parameter is used for the Navier-Stokes fluid solver exclusively.

> **Parameters rho_f** (*float*) – The fluid density [kg/(m*m*m)]

See also: *setFluidViscosity()* and *setFluidCompressibility()*

setFluidPressureModulation(*A*, *f*, *phi=0.0*, *plot=False*)

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) [rad]
>
> - **plot** (*bool*) – Show a plot of the resulting modulation

See also: *setTopWallNormalStressModulation()* and *disableFluidPressureModulation()*

setFluidTopFixedFlux(*specific_flux*)

Define a constant fluid flux normal to the boundary.

The default behavior for the boundary is fixed value (Dirichlet), see *setFluidBottomFixedPressure()*.

> **Parameters specific_flux** – Specific flux values across boundary (positive values upwards), [m/s]

setFluidTopFixedPressure()

Set the upper boundary of the fluid domain to follow the fixed pressure value (Dirichlet) boundary condition.

This is the default behavior for the boundary. See also *setFluidTopNoFlow()*

setFluidTopNoFlow()

Set the upper boundary of the fluid domain to follow the no-flow (Neumann) boundary condition with free slip parallel to the boundary.

The default behavior for the boundary is fixed value (Dirichlet), see *setFluidTopFixedPressure()*.

setFluidTopNoFlowNoSlip()

Set the upper boundary of the fluid domain to follow the no-flow (Neumann) boundary condition with no slip parallel to the boundary.

The default behavior for the boundary is fixed value (Dirichlet), see *setFluidTopFixedPressure()*.

setFluidViscosity(*mu*)

Set the fluid dynamic viscosity [Pa*s]. The value for water is 1.797e-3 at 0 degrees Celcius. This parameter is used for both the Darcy and Navier-Stokes fluid solver.

> **Parameters** **mu** (*float*) – The fluid dynamic viscosity [Pa*s]

See also: ***setFluidDensity()*** and *setFluidCompressibility()*

setFluidXFixedPressure()

Set the X boundaries of the fluid domain to follow the fixed pressure value (Dirichlet) boundary condition.

This is not the default behavior for the boundary. See also *setFluidXFixedPressure()*, *setFluidXNoFlow()*, and *setFluidXPeriodic()* (default)

setFluidXNoFlow()

Set the X boundaries of the fluid domain to follow the no-flow (Neumann) boundary condition.

This is not the default behavior for the boundary. See also *setFluidXFixedPressure()*, *setFluidXNoFlow()*, and *setFluidXPeriodic()* (default)

setFluidXPeriodic()

Set the X boundaries of the fluid domain to follow the periodic (cyclic) boundary condition.

This is the default behavior for the boundary. See also *setFluidXFixedPressure()* and *setFluidXNoFlow()*

setFluidYFixedPressure()

Set the Y boundaries of the fluid domain to follow the fixed pressure value (Dirichlet) boundary condition.

This is not the default behavior for the boundary. See also *setFluidYNoFlow()* and *setFluidYPeriodic()* (default)

setFluidYNoFlow()

Set the Y boundaries of the fluid domain to follow the no-flow (Neumann) boundary condition.

This is not the default behavior for the boundary. See also *setFluidYFixedPressure()* and *setFluidYPeriodic()* (default)

setFluidYPeriodic()

Set the Y boundaries of the fluid domain to follow the periodic (cyclic) boundary condition.

This is the default behavior for the boundary. See also *setFluidYFixedPressure()* and *setFluidYNoFlow()*

setGamma(*gamma*)

Gamma is a fluid solver parameter, used for smoothing the pressure values. The epsilon (pressure) values are smoothed by including the average epsilon value of the six closest (face) neighbor cells. This parameter should be in the range [0.0;1.0[. The higher the value, the more averaging is introduced. A value of 0.0 disables all averaging.

The default and recommended value is 0.0.

> **Parameters** **theta** (*float*) – The smoothing parameter value

Other solver parameter setting functions: *setTheta()*, *setBeta()*, *setTolerance()*, *setDEMstepsPerCFDstep()* and *setMaxIterations()*

setMaxIterations(*maxiter*)

A fluid solver parameter, the value of the maxiter parameter denotes the maximal allowed number of fluid solver iterations before ending the fluid solver loop prematurely. The residual values are at that point not fulfilling the tolerance criteria. The parameter is included to avoid infinite hangs.

The default and recommended value is 1e4.

Parameters **maxiter** (*int*) – The maximum number of Jacobi iterations in the fluid solver

Other solver parameter setting functions: *setGamma()*, *setTheta()*, *setBeta()*, *setDEMstepsPerCFDstep()* and *setTolerance()*

setPermeabilityGrainSize (*verbose=True*)
Set the permeability prefactor based on the mean grain size (Damsgaard et al., 2015, eq. 10).

Parameters **verbose** (*bool*) – Print information about the realistic permeabilities hydraulic conductivities to expect with the chosen permeability prefactor.

setPermeabilityPrefactor (*k_c*, *verbose=True*)
Set the permeability prefactor from Goren et al 2011, eq. 24. The function will print the limits of permeabilities to be simulated. This parameter is only used in the Darcy solver.

Parameters

- **k_c** (*float*) – Permeability prefactor value [m*m]

- **verbose** (*bool*) – Print information about the realistic permeabilities and hydraulic conductivities to expect with the chosen permeability prefactor.

setStaticFriction (*mu_s*)
Set the static friction coefficient for particle-particle interactions (*self.mu_s*). This value describes the resistance to a shearing motion while it is not happenind (contact tangential velocity zero).

Parameters **mu_s** (*float*) – Value of the static friction coefficient, in [0;inf[. Usually between 0 and 1.

See also: setDynamicFriction(mu_d)()

setStiffnessNormal (*k_n*)
Set the elastic stiffness (k_n) in the normal direction of the contact.

Parameters **k_n** (*float*) – The elastic stiffness coefficient [N/m]

setStiffnessTangential (*k_t*)
Set the elastic stiffness (k_t) in the tangential direction of the contact.

Parameters **k_t** (*float*) – The elastic stiffness coefficient [N/m]

setTheta (*theta*)
Theta is a fluid solver under-relaxation parameter, used in solution of Poisson equation. The value should be within the range]0.0;1.0]. At a value of 1.0, the new estimate of epsilon values is used exclusively. At lower values, a linear interpolation between new and old values is used. The solution typically converges faster with a value of 1.0, but instabilities may be avoided with lower values.

The default and recommended value is 1.0.

Parameters **theta** (*float*) – The under-relaxation parameter value

Other solver parameter setting functions: *setGamma()*, *setBeta()*, *setTolerance()*, *setDEMstepsPerCFDstep()* and *setMaxIterations()*

setTolerance (*tolerance*)
A fluid solver parameter, the value of the tolerance parameter denotes the required value of the maximum normalized residual for the fluid solver.

The default and recommended value is 1.0e-3.

Parameters **tolerance** (*float*) – The tolerance criteria for the maximal normalized residual

Other solver parameter setting functions: *setGamma()*, *setTheta()*, *setBeta()*, *setDEMstepsPerCFDstep()* and *setMaxIterations()*

setTopWallNormalStressModulation(*A*, *f*, *plot=False*)

Set the parameters for the sine wave modulating the normal stress at the top wall. Note that a cos-wave is obtained with phi=pi/2.

> **Parameters**
>
> - **A** (*float*) – Fluctuation amplitude [Pa]
>
> - **f** (*float*) – Fluctuation frequency [Hz]
>
> - **plot** (*bool*) – Show a plot of the resulting modulation

See also: *setFluidPressureModulation()* and *disableTopWallNormalStressModulation()*

setYoungsModulus(*E*)

Set the elastic Young's modulus (E) for the contact model. This parameter is used over normal stiffness (k_n) and tangential stiffness (k_t) when its value is greater than zero. Using this parameter produces size-invariant behavior.

Example values are ~70e9 Pa for quartz, http://www.engineeringtoolbox.com/young-modulus-d_417.html

> **Parameters** **E** (*float*) – The elastic modulus [Pa]

shear(*shear_strain_rate=1.0*, *shear_stress=False*)

Setup shear experiment either by a constant shear rate or a constant shear stress. 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 if shear_stress isn't False.
>
> - **shear_stress** (*float or bool*) – The shear stress value to use [Pa].

shearDisplacement()

Calculates and returns the current shear displacement. The displacement is found by determining the total x-axis displacement of the upper, fixed particles.

> **Returns** The total shear displacement [m]
>
> **Return type** float

See also: *shearStrain()* and *shearVelocity()*

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

See also: *shearStrainRate()* and *shearVel()*

shearStrainRate()

Calculates the shear strain rate (dot(gamma)) value of the experiment.

> **Returns** The value of dot(gamma)
>
> **Return type** float

See also: *shearStrain()* and *shearVel()*

shearStress (*type='effective'*)
: Calculates the sum of shear stress values measured on any moving particles with a finite and fixed velocity.

    Parameters type (`str`) – Find the 'defined' or 'effective' (default) shear stress

    Returns The shear stress in Pa

    Return type numpy.array

shearVel ()
: Alias of *shearVelocity()*

shearVelocity ()
: Calculates and returns the current shear velocity. The displacement is found by determining the total x-axis velocity of the upper, fixed particles.

    Returns The shear velocity [m/s]

    Return type float

    See also: *shearStrainRate()* and *shearDisplacement()*

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

show (*coloring=array([], dtype=float64)*, *resolution=6*)
: Show a rendering of all particles in a window.

    Parameters

    - **coloring** (`numpy.array`) – Color the particles from red to white to blue according to the values in this array.

    - **resolution** (`int`) – The resolution of the rendered spheres. Larger values increase the performance requirements.

smallestMass ()
: Returns the mass of the leightest particle.

    Parameters idx (`int`) – Particle index

    Returns The mass of the particle [kg]

    Return type float

staticGrid ()
: Set the height of the fluid grid to be constant as set in *self.L[2]*.

    See also *adaptiveGrid()*

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

surfaceArea (*idx*)
: Returns the surface area of a particle.

    Parameters idx (`int`) – Particle index

    Returns The surface area of the particle [m^2]

> > > **Return type** float

thinsection_x1x3 (*x2='center'*, *graphics_format='png'*, *cbmax=None*, *arrowscale=0.01*, *velar-rowscale=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*

torqueScriptPenguin (*email='adc@geo.au.dk'*, *email_alerts='ae'*, *walltime='1920:00:00'*, *queue='H30G'*, *spheredir='/home/adc/code/sphere'*)

> 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

- **spheredir** (*str*) – The path to the root directory of sphere on the cluster

totalFrictionalEnergy ()

Returns the total frictional dissipated energy for all particles.

> **Returns** The total frictional energy lost of all particles [J]
>
> **Return type** float

totalKineticEnergy ()

Returns the total linear kinetic energy for all particles.

> **Returns** The kinetic energy of all particles [J]

totalMass ()

Returns the total mass of all particles.

> **Returns** The total mass in [kg]

totalMomentum ()

Returns the sum of particle momentums.

> **Returns** The sum of particle momentums (m*v) [N*s]
>
> **Return type** numpy.array

totalRotationalEnergy ()

Returns the total rotational kinetic energy for all particles.

> **Returns** The rotational energy of all particles [J]

totalViscousEnergy ()

Returns the total viscous dissipated energy for all particles.

> **Returns** The normal viscous energy lost by all particles [J]
>
> **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 *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

viscousEnergy(*idx*)

Returns the viscous dissipated energy for a particle.

> **Parameters idx** (`int`) – Particle index
>
> **Returns** The energy lost by the particle by viscous dissipation [J]
>
> **Return type** float

visualize(*method='energy'*, *savefig=True*, *outformat='png'*, *figsize=False*, *pickle=False*, *xlim=False*, *firststep=0*, *f_min=None*, *f_max=None*, *cmap=None*)

Visualize output from the simulation, 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 simulation, and the visualization method.

Parameters

- **method** (`str`) – The type of plot to render. Possible values are 'energy', 'walls', 'triaxial', 'inertia', 'mean-fluid-pressure', 'fluid-pressure', 'shear', 'shear-displacement', 'porosity', 'rate-dependence', 'contacts'
- **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').
- **figsize** (`array`) – Specify output figure size in inches
- **pickle** (`bool`) – Save all figure content as a Python pickle file. It can be opened later using *fig = pickle.load(open('file.pickle','rb'))*.
- **xlim** (`array`) – Set custom limits to the x axis. If not specified, the x range will correspond to the entire data interval.
- **firststep** (`int`) – The first output file step to read (default: 0)
- **cmap** (`matplotlib.colors.LinearSegmentedColormap`) – Choose custom color map, e.g. *cmap=matplotlib.cm.get_cmap('afmhot')*

voidRatio()

Calculates the current void ratio

> **Returns** The void ratio, in [0:1]
>
> **Return type** float

volume(*idx*)

Returns the volume of a particle.

> **Parameters idx** (`int`) – Particle index

>
> > **Returns** The volume of the particle [m^3]
> >
> > **Return type** float

wall0iz()
> Returns the cell index of wall 0 along z.
>
> > **Returns** z cell index
> >
> > **Return type** int

wet()
> Set the simulation to be wet (total fluid saturation).
>
> See also *dry()*

writeFluidVTK(*folder='../output/'*, *cell_centered=True*, *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 scalars (pressure, porosity, porosity change) and the velocity vectors are either placed in a grid where the grid corners correspond to the computational grid center (cell_centered = False). This results in a grid that doesn't appears to span the simulation domain, and values are smoothly interpolated on the cell faces. Alternatively, the visualization grid is equal to the computational grid, and cells face colors are not interpolated (cell_centered = True, default behavior).
>
> 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.
>
> To visualize the cell-centered data with smooth interpolation, and in order to visualize fluid vector fields, the cell-centered mesh is selected in the "Pipeline Browser", and is filtered using "Filters" -> "Alphabetical" -> "Cell Data to Point Data".
>
> 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/')
> >
> > - **cell_centered** (*bool*) – put scalars and vectors at cell centers (True) or cell corners (False), (default = True)
> >
> > - **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", 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 `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(*cell_centered=True*, *verbose=True*, *forces=False*)

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)
> - **cell_centered** (`bool`) – Write fluid values to cell centered positions (default = true)
> - **forces** (`bool`) – Write contact force files (slow) (default = False)

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

> Writes a VTK file with particle-interaction information to the `../output/` folder by default. The file name will be in the format `<self.sid>.vtp`. The vtp files can be used to visualize the particle interactions in ParaView. First use the "Cell Data to Point Data" filter, and afterwards show the contact network with the "Tube" filter.
>
> > **Parameters**
> >
> > - **folder** (*str*) – The folder where to place the output file (default (default = '../output/')
> >
> > - **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.**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 *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

1.5 sphere internals

The *sphere* executable has the following options:

```
$ ../../sphere --help
../../sphere: particle dynamics simulator
Usage: ../../sphere [OPTION[S]]... [FILE1 ...]
Options:
-h, --help               print help
-V, --version              print version information and exit
-q, --quiet              suppress status messages to stdout
-d <device>              execute on device with specified id
-n, --dry            show key experiment parameters and quit
-f, --fluid            simulate fluid between particles
-r, --render             render input files to images instead of
                     simulating the temporal evolution
```

```
-dc, --dont-check        don't check values before running

Raytracer (-r) specific options:
-m <method> <maxval> [-l <lower cutoff val>], or
--method <method> <maxval> [-l <lower cutoff val>]
        color visualization method, possible values:
        normal, pres, vel, angvel, xdisp, angpos
        'normal' is the default mode
        if -l is appended, don't render particles with value below
-c, --contacts                  Print a list of particle-particle contacts
```

The most common way to invoke *sphere* is however via the Python API (e.g. `sphere.run()`, `sphere.render()`, etc.).

subsection{The *sphere* algorithm} label{subsec:spherealgo} The *sphere*-binary is launched from the system terminal by passing the simulation ID as an input parameter; texttt{./sphere_<architecture> <simulation_ID>}. The sequence of events in the program is the following: #. System check, including search for NVIDIA CUDA compatible devices (texttt{main.cpp}).

1. Initial data import from binary input file (texttt{main.cpp}).

2. Allocation of memory for all host variables (particles, grid, walls, etc.) (texttt{main.cpp}).

3. Continued import from binary input file (texttt{main.cpp}).

4. Control handed to GPU-specific function texttt{gpuMain(ldots)} (texttt{device.cu}).

5. Memory allocation of device memory (texttt{device.cu}).

6. Transfer of data from host to device variables (texttt{device.cu}).

7. Initialization of Thrustfootnote{url{https://code.google.com/p/thrust/}} radix sort configuration (texttt{device.cu}).

8. Calculation of GPU workload configuration (thread and block layout) (texttt{device.cu}).

9. Status and data written to verb"<simulation_ID>.status.dat" and verb"<simulation_ID>.output0.bin", both located in texttt{output/} folder (texttt{device.cu}).

10. Main loop (while texttt{time.current <= time.total}) (functions called in texttt{device.cu}, function definitions in seperate files). Each kernel call is wrapped in profiling- and error exception handling functions:

1. label{loopstart}CUDA thread synchronization point.

2. texttt{calcParticleCellID<<<,>>>(ldots)}: Particle-grid hash value calculation (texttt{sorting.cuh}).

3. CUDA thread synchronization point.

4. texttt{thrust::sort_by_key(ldots)}: Thrust radix sort of particle-grid hash array (texttt{device.cu}).

5. texttt{cudaMemset(ldots)}: Writing zero value (texttt{0xffffffff}) to empty grid cells (texttt{device.cu}).

6. texttt{reorderArrays<<<,>>>(ldots)}: Reordering of particle arrays, based on sorted particle-grid-hash values (texttt{sorting.cuh}).

7. CUDA thread synchronization point.

8. Optional: texttt{topology<<<,>>>(ldots)}: If particle contact history is required by the contact model, particle contacts are identified, and stored per particle. Previous, now non-existant contacts are discarded (texttt{contactsearch.cuh}).

9. CUDA thread synchronization point.

10. texttt{interact<<<,>>>(ldots)}: For each particle: Search of contacts in neighbor cells, processing of optional collisions and updating of resulting forces and torques. Values are written to read/write device memory arrays (texttt{contactsearch.cuh}).

11. CUDA thread synchronization point.

12. texttt{integrate<<<,>>>(ldots)}: Updating of spatial degrees of freedom by a second-order Taylor series expansion integration (texttt{integration.cuh}).

13. CUDA thread synchronization point.

14. texttt{summation<<<,>>>(ldots)}: Particle contributions to the net force on the walls are summated (texttt{integration.cuh}).

15. CUDA thread synchronization point.

16. texttt{integrateWalls<<<,>>>(ldots)}: Updating of spatial degrees of freedom of walls (texttt{integration.cuh}).

17. Update of timers and loop-related counters (e.g. texttt{time.current}), (texttt{device.cu}).

18. If file output interval is reached:

    > item Optional write of data to output binary (verb"<simulation_ID>.output#..bin"), (texttt{file_io.cpp}). item Update of verb"<simulation_ID>.status#..bin" (texttt{device.cu}).

    item Return to point ref{loopstart}, unless texttt{time.current >= time.total}, in which case the program continues to point ref{loopend}.

1. label{loopend}Liberation of device memory (texttt{device.cu}).

2. Control returned to texttt{main(ldots)}, liberation of host memory (texttt{main.cpp}).

3. End of program, return status equal to zero (0) if no problems where encountered.

1.5.1 Numerical algorithm

The *sphere*-binary is launched from the system terminal by passing the simulation ID as an input parameter; texttt{./sphere_<architecture> <simulation_ID>}. The sequence of events in the program is the following:

1. System check, including search for NVIDIA CUDA compatible devices (texttt{main.cpp}).

2. Initial data import from binary input file (texttt{main.cpp}).

3. Allocation of memory for all host variables (particles, grid, walls, etc.) (texttt{main.cpp}).

4. Continued import from binary input file (texttt{main.cpp}).

5. Control handed to GPU-specific function texttt{gpuMain(ldots)} (texttt{device.cu}).

6. Memory allocation of device memory (texttt{device.cu}).

7. Transfer of data from host to device variables (texttt{device.cu}).

8. Initialization of Thrustfootnote{url{https://code.google.com/p/thrust/}} radix sort configuration (texttt{device.cu}).

9. Calculation of GPU workload configuration (thread and block layout) (texttt{device.cu}).

10. Status and data written to verb"<simulation_ID>.status.dat" and verb"<simulation_ID>.output0.bin", both located in texttt{output/} folder (texttt{device.cu}).

11. Main loop (while texttt{time.current <= time.total}) (functions called in texttt{device.cu}, function definitions in seperate files). Each kernel call is wrapped in profiling- and error exception handling functions:

1. label{loopstart}CUDA thread synchronization point.

---

2. texttt{calcParticleCellID<<<,>>>(ldots)}: Particle-grid hash value calculation (texttt{sorting.cuh}).

3. CUDA thread synchronization point.

4. texttt{thrust::sort_by_key(ldots)}: Thrust radix sort of particle-grid hash array (texttt{device.cu}).

5. texttt{cudaMemset(ldots)}: Writing zero value (texttt{0xffffffff}) to empty grid cells (texttt{device.cu}).

6. texttt{reorderArrays<<<,>>>(ldots)}: Reordering of particle arrays, based on sorted particle-grid-hash values (texttt{sorting.cuh}).

7. CUDA thread synchronization point.

8. Optional: texttt{topology<<<,>>>(ldots)}: If particle contact history is required by the contact model, particle contacts are identified, and stored per particle. Previous, now non-existant contacts are discarded (texttt{contactsearch.cuh}).

9. CUDA thread synchronization point.

10. texttt{interact<<<,>>>(ldots)}: For each particle: Search of contacts in neighbor cells, processing of optional collisions and updating of resulting forces and torques. Values are written to read/write device memory arrays (texttt{contactsearch.cuh}).

11. CUDA thread synchronization point.

12. texttt{integrate<<<,>>>(ldots)}: Updating of spatial degrees of freedom by a second-order Taylor series expansion integration (texttt{integration.cuh}).

13. CUDA thread synchronization point.

14. texttt{summation<<<,>>>(ldots)}: Particle contributions to the net force on the walls are summated (texttt{integration.cuh}).

15. CUDA thread synchronization point.

16. texttt{integrateWalls<<<,>>>(ldots)}: Updating of spatial degrees of freedom of walls (texttt{integration.cuh}).

17. Update of timers and loop-related counters (e.g. texttt{time.current}), (texttt{device.cu}).

18. If file output interval is reached:

    • Optional write of data to output binary (verb"<simulation_ID>.output#..bin"), (texttt{file_io.cpp}).

    • Update of verb"<simulation_ID>.status#..bin" (texttt{device.cu}).

19. Return to point ref{loopstart}, unless texttt{time.current >= time.total}, in which case the program continues to point ref{loopend}.

1. label{loopend}Liberation of device memory (texttt{device.cu}).

2. Control returned to texttt{main(ldots)}, liberation of host memory (texttt{main.cpp}).

3. End of program, return status equal to zero (0) if no problems where encountered.

The length of the computational time steps (texttt{time.dt}) is calculated via equation ref{eq:dt}, where length of the time intervals is defined by:

$$\Delta t = 0.075 \min \left(m / \max(k_n, k_t) \right)$$

where m is the particle mass, and k are the elastic stiffnesses. The time step is set by this relationship in initTemporal(). This equation ensures that the elastic wave (traveling at the speed of sound) is resolved a number of times while traveling through the smallest particle.

subsubsection{Host and device memory types} label{subsubsec:memorytypes} A full, listed description of the *sphere* source code variables can be found in appendix ref{apx:SourceCodeVariables}, page

pageref{apx:SourceCodeVariables}. There are three types of memory types employed in the *sphere* source code, with different characteristics and physical placement in the system (figure ref{fig:memory}).

The floating point precision operating internally in *sphere* is defined in texttt{datatypes.h}, and can be either single (texttt{float}), or double (texttt{double}). Depending on the GPU, the calculations are performed about double as fast in single precision, in relation to double precision. In dense granular configuraions, the double precision however results in greatly improved numerical stability, and is thus set as the default floating point precision. The floating point precision is stored as the type definitions texttt{Float}, texttt{Float3} and texttt{Float4}. The floating point values in the in- and output datafiles are emph{always} written in double precision, and, if necessary, automatically converted by *sphere*.

Three-dimensional variables (e.g. spatial vectors in *E^3*) are in global memory stored as texttt{Float4} arrays, since these read and writes can be coalesced, while e.g. texttt{float3}'s cannot. This alone yields a *sim'20*times' performance boost, even though it involves 25% more (unused) data.

paragraph{Host memory} is the main random-access computer memory (RAM), i.e. read and write memory accessible by CPU processes, but inaccessible by CUDA kernels executed on the device.

paragraph{Device memory} is the main, global device memory. It resides off-chip on the GPU, often in the form of 1–6 GB DRAM. The read/write access from the CUDA kernels is relatively slow. The arrays residing in (global) device memory are prefixed by dev_ in the source code.

marginpar{Todo: Expand section on device memory types}

paragraph{Constant memory} values cannot be changed after they are set, and are used for scalars or small vectors. Values are set in the transferToConstantMemory(...)} function, called in the beginning of texttt{gpuMain(ldots)} in texttt{device.cu}. Constant memory variables have a global scope, and are prefixed by devC_ in the source code.

%subsection{The main loop} %label{subsec:mainloop} %The *sphere* software calculates particle movement and rotation based on the forces applied to it, by application of Newton's law of motion (Newton's second law with constant particle mass: *F_{mathrm{net}} = m cdot a_{mathrm{cm}}*). This is done in a series of algorithmic steps, see list on page pageref{loopstart}. The steps are explained in the following sections with reference to the *sphere*-source file; texttt{sphere.cu}. The intent with this document is emph{not} to give a full theoretical background of the methods, but rather how the software performs the calculations.

subsection{Performance} marginpar{Todo: insert graph of performance vs. np and performance vs. *Delta t*}. subsubsection{Particles and computational time}

subsection{Compilation} label{subsec:compilation} An important note is that the texttt{C} examples of the NVIDIA CUDA SDK should be compiled before *sphere*. Consult the *Getting started guide*, supplied by Nvidia for details on this step.

sphere is supplied with several Makefiles, which automate the compilation process. To compile all components, open a shell, go to the texttt{src/} subfolder and type texttt{make}. The GNU Make will return the parameters passed to the individual CUDA and GNU compilers (texttt{nvcc} and texttt{gcc}). The resulting binary file (texttt{sphere}) is placed in the *sphere* root folder. src/Makefile will also compile the raytracer.

1.5.2 C++ reference

class DEM

INDICES AND TABLES

- genindex
- search

S

A

B

C

D

E

F

G

H

I

K

L