sphere Documentation

Release 0.35

Anders Damsgaard

January 25, 2014

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 Lunbek Egholm and Jan A. Piotrowski, all of the Department of Geoscience, Aarhus University, Denmark. The author welcomes interested third party developers. This document is a work in progress.

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

CHAPTER

ONE

CONTENTS

1.1 Introduction

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

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

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

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

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

1.1.1 Requirements

The build requirements are:

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

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

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

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

The runtime requirements are:

- A CUDA-enabled GPU with compute capability 1.1 or greater.

- A Nvidia CUDA-enabled GPU and device driver

Optional tools, required for simulation setup and data processing:

- Python 2.7
- 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
```

Optional tools, required for building the documentation:

- Sphinx
  - sphinxcontrib-programoutput
- Doxygen
- Breathe
- dvipng

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

```
$ sudo apt-get install python-sphinx python-pip doxygen dvipng python-sphinxcontrib-programoutput tex
$ sudo pip install breathe
```

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

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

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

1.1.2 Obtaining sphere

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

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

1.1.3 Building `sphere`

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

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

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

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

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

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

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

```
$ make test
```

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

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

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

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

The output should look similar to this:

Command u'../../sphere –version' failed: [Errno 2] No such file or directory

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

```
$ make test
```

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

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

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

To see all available output formats, execute:

```
$ make help
```

1.1.4 Updating sphere

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

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

1.1.5 Work flow

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

- Setup of particle assemblage, physical properties and conditions using the Python API.

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

1.2 Discrete element method

The discrete element method (or distinct element method) was initially formulated by Cundall and Strack (1979). It simulates the physical behavior and interaction of discrete, unbreakable particles, with their own mass and inertia, under the influence of e.g. gravity and boundary conditions such as moving walls. By discretizing time into small time steps, explicit integration of Newton's second law of motion is used to predict the new position and kinematic values for each particle from the previous sums of forces. This Lagrangian approach is ideal for simulating discontinuous materials, such as granular matter. The complexity of the computations is kept low by representing the particles as spheres, which keeps contact-searching algorithms simple.

1.3 Fluid simulation and particle-fluid interaction

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

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

1.3.1 Derivation of the Navier Stokes equations with porosity

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

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

and the momentum equation:

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

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

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

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

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

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:

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

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

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

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

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

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

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

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

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

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

Spatial variations in the porosity are neglected,

$$\nabla \phi := 0$$

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

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

With these assumptions, the momentum equation simplifies to:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

—

1.3.3 Solution procedure by operator splitting

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

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

$$\Downarrow$$

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

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

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

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

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

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

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

The divergence of a scalar and vector can be split:

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

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

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

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

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

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

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

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

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

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

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

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

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

$$f = f_1 - f_2$$

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

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

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

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

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

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

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

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

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

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

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

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

1.3.4 Numerical implementation

—

1.4 Python API

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

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

1.4.1 Sample usage

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

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

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


### SIMULATION SETUP

# Create a sphere object with two preallocated particles and a simulation ID
SB = sphere.Spherebin(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

# let's disable gravity in this simulation
GRAVITY = numpy.array([0.0, 0.0, 0.0])

# 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(g = GRAVITY, margin = 5.0)

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

```

```
# Save the simulation as a input file for sphere
SB.writebin()

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


### RUNNING THE SIMULATION

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


### ANALYSIS OF SIMULATION RESULTS

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

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

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

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

1.4.2 The `sphere` module

class `sphere.`**`Spherebin`**(*np=1*, *nd=3*, *nw=1*, *sid='unnamed'*, *fluid=False*)

Class containing all `sphere` data.

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

Parameters

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

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

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

Parameters

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

adjustUpperWall(*z_adjust=1.1*)
Included for legacy purposes, calls adjustWall with idx=0

adjustWall(*idx*, *adjust=1.1*)
Adjust grid and dynamic wall to max. particle position

bond(*i*, *j*)
Create a bond between particles i and j

bondsRose(*imgformat='pdf'*)
Visualize strike- and dip angles of the bond pairs in a rose plot.

bulkPorosity()
Calculate and return the bulk porosity

consolidate(*deviatoric_stress=10000.0*, *periodic=1*)
Setup consolidation experiment. Specify the upper wall deviatoric stress in Pascal, default value is 10 kPa.

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

currentDevs()
Return current magnitude of the deviatoric normal stress

defaultParams(*mu_s=0.4*, *mu_d=0.4*, *mu_r=0.0*, *rho=2600*, *k_n=1160000000.0*, *k_t=1160000000.0*, *k_r=0*, *gamma_n=0*, *gamma_t=0*, *gamma_r=0*, *gamma_wn=10000.0*, *gamma_wt=10000.0*, *capillaryCohesion=0*, *nu=0.0*)
Initialize particle parameters to default values. Radii must be set prior to calling this function.

energy(*method*)
Calculate the sum of the energy components of all particles.

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. @param lc: Lower cutoff of contact forces. Contacts below are not visualized (float) @param uc: Upper cutoff of contact forces. Contacts above are visualized with this value (float) @param outformat: Format of output image. Possible values are 'interactive', 'png', 'epslatex', 'epslatex-color' @param disp: Display forcechains in '2d' or '3d' Warning: Will segfault if no contacts are found.

forcechainsRose(*lower_limit=0.25*)
Visualize strike- and dip angles of the strongest force chains in a rose plot.

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).

generateRadii(*psd='logn'*, *radius_mean=0.00044*, *radius_variance=8.8e-09*, *histogram=True*)
Draw random particle radii from a selected probability distribution. The larger the variance of radii is, the slower the computations will run. The reason is two-fold: The smallest particle dictates the time step length, where smaller particles cause shorter time steps. At the same time, the largest particle determines the sorting cell size, where larger particles cause larger cells. Larger cells are likely to contain more particles, causing more contact checks.

Parameters

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

See also: `generateBimodalRadii()`.

initFluid(*nu=0.00089*)
Initialize the fluid arrays and the fluid viscosity

initGrid()
Initialize grid suitable for the particle positions set previously. The margin parameter adjusts the distance (in no. of max. radii) from the particle boundaries.

initGridAndWorldsize(*g=array([0., 0., -9.80665])*, *margin=2.0*, *periodic=1*, *contactmodel=2*)
Initialize grid suitable for the particle positions set previously. The margin parameter adjusts the distance (in no. of max. radii) from the particle boundaries.

initGridPos(*g=array([0., 0., -9.80665])*, *gridnum=array([12, 12, 36])*, *periodic=1*, *contactmodel=2*)
Initialize particle positions in loose, cubic configuration. Radii must be set beforehand. xynum is the number of rows in both x- and y- directions.

initRandomGridPos(*g=array([0., 0., -9.80665])*, *gridnum=array([12, 12, 32])*, *periodic=1*, *contactmodel=2*)
Initialize particle positions in loose, cubic configuration. Radii must be set beforehand. xynum is the number of rows in both x- and y- directions.

initRandomPos(*g=array([0., 0., -9.80665])*, *gridnum=array([12, 12, 36])*, *periodic=1*, *contactmodel=2*)
Initialize particle positions in loose, cubic configuration. Radii must be set beforehand. xynum is the number of rows in both x- and y- directions.

initTemporal(*total*, *current=0.0*, *file_dt=0.05*, *step_count=0*)
Set temporal parameters for the simulation. Particle radii and physical parameters need to be set prior to these.

plotConvergence(*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.<format>.

Parameters **format** (*str*) – The plot file type (default = 'png')

plotFluidDiffAdvPresZ()
Compare contributions to the velocity from diffusion and advection at top boundary, 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.

plotFluidPorositiesY(*iteration=-1*, *y=-1*, *outformat='png'*)
Plot the porosity values from the simulation. If iteration is -1 (default value), the last output file will be shown. If the y value is -1, the center x,z plane will be rendered

plotPrescribedFluidPressures(*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.

porosities(*outformat='pdf'*, *zslices=16*)
Plot porosities with depth

porosity(*slices=10*, *verbose=False*)
Calculate the porosity as a function of depth, by averaging values in horizontal slabs. Returns porosity values and depth

random2bonds(*ratio=0.3*, *spacing=-0.1*)
Bond an amount of particles in two-particle clusters @param ratio: The amount of particles to bond, values in]0.0;1.0] (float) @param spacing: The distance relative to the sum of radii between bonded particles, neg. values denote an overlap. Values in]0.0,inf[(float). 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.

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

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

Parameters

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

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

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

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

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

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

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

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

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

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

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

Parameters

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

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

render(*method='pres'*, *max_val=1000.0*, *lower_cutoff=0.0*, *graphicsformat='png'*, *verbose=True*)
Render all output files that belong to the simulation, determined by sid.

run(*verbose=True*, *hideinputfile=False*, *dry=False*, *valgrind=False*, *cudamemcheck=False*)
Execute `sphere` with target project

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

Parameters

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

shear(*shear_strain_rate=1*, *periodic=1*)
Setup shear experiment. Specify the upper wall deviatoric stress in Pascal, default value is 10 kPa. The shear strain rate is the shear length divided by the initial height per second.

sheardisp(*outformat='pdf'*, *zslices=32*)
Show particle x-displacement vs. the z-pos

shearstrain()
Calculates and returns the current shear strain (gamma) value of the experiment.

shearvel()
Calculates and returns the shear velocity (gamma_dot) of the experiment

status()
Show the current simulation status

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

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

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

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'*, *workdir='/scratch'*)
Create job script for the Torque queue manager for the binary

triaxial (*wvel=-0.001*, *deviatoric_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 deviatoric stress

uniaxialStrainRate (*wvel=-0.001*, *periodic=1*)
Setup consolidation experiment. Specify the upper wall velocity in m/s, default value is -0.001 m/s (i.e. downwards).

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

voidRatio ()
Returns the current void ratio

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

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

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

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

Parameters

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

writeVTK (*folder='../output/'*, *verbose=True*)
Writes a VTK file with particle information to the `../output/` folder by default. The file name will be in the format `<self.sid>.vtu`. The vtu files can be used to visualize the particles in ParaView.

After opening the vtu files, the particle fields will show up in the "Properties" list. Press "Apply" to import all fields into the ParaView session. The particles are visualized by selecting the imported data in the "Pipeline Browser". Afterwards, click the "Glyph" button in the "Common" toolbar, or go to the "Filters" menu, and press "Glyph" from the "Common" list. Choose "Sphere" as the "Glyph Type", set "Radius"

to 1.0, choose "scalar" as the "Scale Mode". Check the "Edit" checkbox, and set the "Set Scale Factor" to 1.0. The field "Maximum Number of Points" may be increased if the number of particles exceed the default value. Finally press "Apply", and the particles will appear in the main window.

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

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

Parameters

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

writeVTKall(*verbose=True*)

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

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

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

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

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

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

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

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

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

See also `readbin()`.

Parameters

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

zeroKinematics()
Zero kinematics of particles

sphere.**V_sphere**(*r*)
Returns the volume of a sphere with radius r

sphere.**cleanup**(*spherebin*)
Remove input/output files and images from simulation

sphere.**convert**(*graphicsformat='png'*, *folder='../img_out'*)
Converts all PPM images in img_out to graphicsformat, using ImageMagick

sphere.**render**(*binary*, *method='pres'*, *max_val=1000.0*, *lower_cutoff=0.0*, *graphicsformat='png'*, *verbose=True*)
Render target binary using the sphere raytracer.

sphere.**run**(*binary*, *verbose=True*, *hideinputfile=False*)
Execute sphere with target binary as input

sphere.**status**(*project*)
Check the status.dat file for the target project, and return the last file numer.

sphere.**thinsectionVideo**(*project*, *out_folder='./'*, *video_format='mp4'*, *fps=25*, *qscale=1*, *bitrate=1800*, *verbose=False*)
Use ffmpeg to combine thin section images to animation. This function will start off by rendering the images.

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'*, *workdir='/scratch'*)
Create job script for the Torque queue manager for three binaries, executed in parallel. Returns the filename of the script

sphere.**torqueScriptSerial3**(*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'*, *workdir='/scratch'*)
Create job script for the Torque queue manager for three binaries

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

sphere.**visualize**(*project*, *method='energy'*, *savefig=True*, *outformat='png'*)
Visualize output from the target project, where the temporal progress is of interest.

1.5 sphere internals

The *sphere* executable has the following options:

Command u'../../sphere –help' failed: [Errno 2] No such file or directory

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**

Public Functions

DEM(std::string inputbin, const int verbosity = 1, const int checkVals = 1, const int dry = 0, const int initCuda = 1, const int transferConstMem = 1, const int darcyflow = 0)

~DEM(void)

void **readbin**(const char * target)

void **writebin**(const char * target)

void **checkValues**(void)

void **reportValues**(void)

void **startTime**(void)

void **render**(const int method = 1, const float maxval = 1.0e3f, const float lower_cutoff = 0.0f, const float focalLength = 1.0f, const unsigned int img_width = 800, const unsigned int img_height = 800)

void **writePPM**(const char * target)

void **porosity**(const int z_slices = 10)

Float3 **minPos**(void)

Float3 **maxPos**(void)

void **findOverlaps**(std::vector< std::vector< unsigned int > > & ij, std::vector< Float > & delta_n_ij)

void **forcechains**(const std::string format = "interactive", const int threedim = 1, const double lower_cutoff = 0.0, const double upper_cutoff = 1.0e9)

void **printNSarray**(FILE * stream, Float * arr)

void **printNSarray**(FILE * stream, Float * arr, std::string desc)

void **printNSarray**(FILE * stream, Float3 * arr)

void **printNSarray**(FILE * stream, Float3 * arr, std::string desc)

void **writeNSarray**(Float * array, const char * filename)

void **writeNSarray**(Float3 * array, const char * filename)

CHAPTER

TWO

INDICES AND TABLES

- *genindex*
- *search*

S

V

W

Z