Users guide to SPHERE: GPU based discrete element method software

Anders Damsgaard Christensen
adc@geo.au.dk
http://cs.au.dk/~adc/

Last revision: September 14, 2012

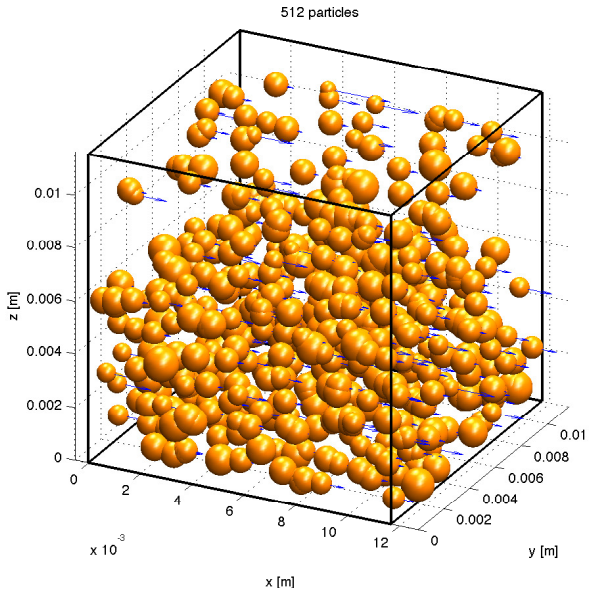
Version **0.2**

Abstract

This document is the official documentation for the SPHERE discrete element modelling software. It presents the theory behind the discrete element method (DEM), the structure of the software source code, and the Python API for handling simulation setup and data analysis.

SPHERE is developed by Anders Damsgaard Christensen under supervision of David Lunbek Egholm and Jan A. Piotrowski, all of the department of Geology, University of Aarhus, Denmark. This document is a work in progress, and is still in an early, unfinished state. This document is typeset with LaTeX 2$_\varepsilon$, with a wide margin (LaTeX standard) to make space for handwritten notes.

Todo: Change cover image



Revision history

Date	Doc. version	SPHERE version	Description
2010-12-06	0.1	Beta 0.03	Initial draft
2012-09-13	0.2	Beta 0.25	Updated for Python API and major source code changes

Contents

1 Introduction

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

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 requirements to the host computer are:

- UNIX, Linux or Mac OS X operating system.
- GCC, the GNU compiler collection.
- A CUDA-enabled GPU with compute capability 1.1 or greater[1].
- The CUDA Developer Drivers and the CUDA Toolkit[2].

For simulation setup and data handling, a Python distribution of a recent version is essential. Required Python modules include Numpy[3]. There is however no requirement of Python on the computer running the SPHERE calculations, i.e. model setup and data analysis can be performed on a separate device. Command examples in this document starting with the symbol '$' are executed in the shell of the operational system, and '>>>' means execution in Python. All numerical values in this document, the source code, and the configuration files are typeset with strict respect to the SI unit system.

2 Discrete element method theory

The discrete element method (or distinct element method) was initially formulated by **?**. 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 ($\Delta t \approx 10^{-8}$s), 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 granularities. The complexity of the computations is kept low by representing the particles as spheres, which keeps contact-searching algorithms simple.

Todo: Expand this section; contact models, etc.

3 SPHERE source code structure

The source code is located in the sphere/src/ folder. After compiling the SPHERE binary (see sub-section 3.3), the procedure of a creating and handling a simulation is typically arranged in the following order:

[1] See http://www.nvidia.com/object/cuda_gpus.html for an official list of NVIDIA CUDA GPUs.

[2] Obtainable free of charge from http://developer.nvidia.com/object/cuda_3_2_downloads.html

[3] http://numpy.scipy.org

1. Setup of particle assemblage, physical properties and conditions using the Python API, described in section 4, page 8.
2. 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. Described in section 5, page 10.
3. Inspection, analysis, interpretation and visualization of SPHERE output in Python. Described in section 6, page 10.

3.1 The SPHERE algorithm

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

1. System check, including search for NVIDIA CUDA compatible devices (`main.cpp`).
2. Initial data import from binary input file (`main.cpp`).
3. Allocation of memory for all host variables (particles, grid, walls, etc.) (`main.cpp`).
4. Continued import from binary input file (`main.cpp`).
5. Control handed to GPU-specific function `gpuMain(...)` (`device.cu`).
6. Memory allocation of device memory (`device.cu`).
7. Transfer of data from host to device variables (`device.cu`).
8. Initialization of Thrust[4] radix sort configuration (`device.cu`).
9. Calculation of GPU workload configuration (thread and block layout) (`device.cu`).
10. Status and data written to `<simulation_ID>.status.dat` and `<simulation_ID>.output0.bin`, both located in `output/` folder (`device.cu`).
11. Main loop (while `time.current <= time.total`) (functions called in `device.cu`, function definitions in seperate files). Each kernel call is wrapped in profiling- and error exception handling functions:
    (a) CUDA thread synchronization point.
    (b) `calcParticleCellID<<<,>>>(...)`: Particle-grid hash value calculation (`sorting.cuh`).
    (c) CUDA thread synchronization point.
    (d) `thrust::sort_by_key(...)`: Thrust radix sort of particle-grid hash array (`device.cu`).
    (e) `cudaMemset(...)`: Writing zero value (`0xffffffff`) to empty grid cells (`device.cu`).

[4] https://code.google.com/p/thrust/

(f) `reorderArrays<<<,>>>(...)`: Reordering of particle arrays, based on sorted particle-grid-hash values (`sorting.cuh`).

(g) CUDA thread synchronization point.

(h) Optional: `topology<<<,>>>(...)`: 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 (`contactsearch.cuh`).

(i) CUDA thread synchronization point.

(j) `interact<<<,>>>(...)`: 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 (`contactsearch.cuh`).

(k) CUDA thread synchronization point.

(l) `integrate<<<,>>>(...)`: Updating of spatial degrees of freedom by a second-order Taylor series expansion integration (`integration.cuh`).

(m) CUDA thread synchronization point.

(n) `summation<<<,>>>(...)`: Particle contributions to the net force on the walls are summated (`integration.cuh`).

(o) CUDA thread synchronization point.

(p) `integrateWalls<<<,>>>(...)`: Updating of spatial degrees of freedom of walls (`integration.cuh`).

(q) Update of timers and loop-related counters (e.g. `time.current`), (`device.cu`).

(r) If file output interval is reached:

i. Optional write of data to output binary (`<simulation_ID>.output#.bin`), (`file_io.cpp`).

ii. Update of `<simulation_ID>.status#.bin` (`device.cu`).

(s) Return to point 11a, unless `time.current >= time.total`, in which case the program continues to point 12.

12. Liberation of device memory (`device.cu`).

13. Control returned to `main(...)`, liberation of host memory (`main.cpp`).

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

The length of the computational time steps (`time.dt`) is calculated via equation 1, where length of the time intervals is defined by:

$$\Delta t = 0.17 \min\left(m / \max(k_n, k_t)\right) \tag{1}$$

where m is the particle mass, and k are the elastic stiffnesses. 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.

3.1.1 Host and device memory types

A full, listed description of the SPHERE source code variables can be found in appendix B, page 15. There are three types of memory types employed in the SPHERE source code, with different characteristics and physical placement in the system (figure ??).

The floating point precision operating internally in SPHERE is defined in datatypes.h, and can be either single (float), or double (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 Float, Float3 and Float4. The floating point values in the in- and output datafiles are *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 Float4 arrays, since these read and writes can be coalesced, while e.g. float3's cannot. This alone yields a ∼20× performance boost, even though it involves 25% more (unused) data.

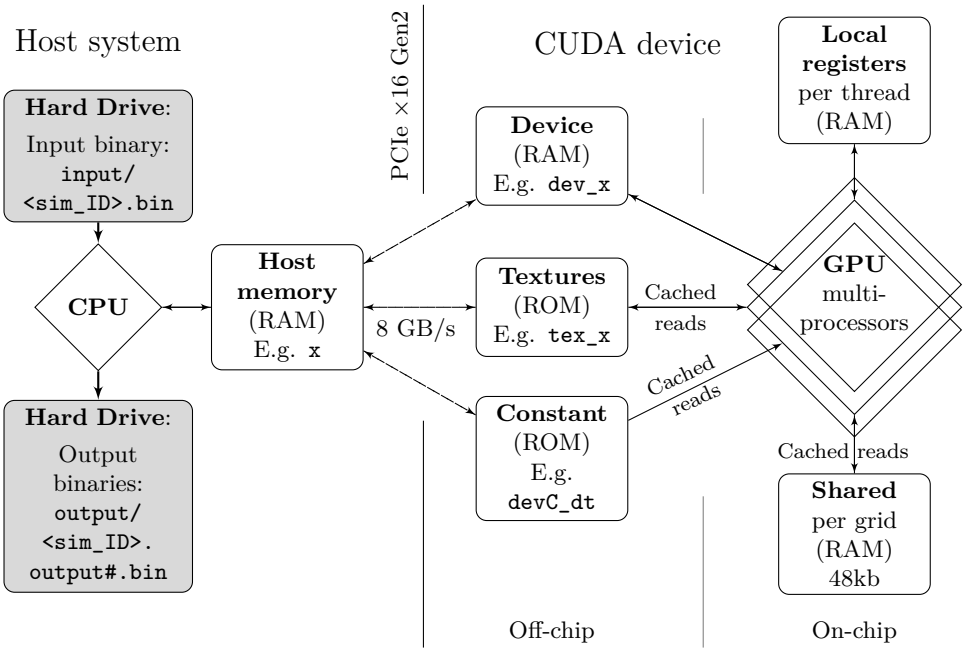


Figure 1: Flow chart of system memory types and communication paths. RAM: Random-Access Memory (read + write), ROM: Read-Only Memory. Specified communication path bandwidth on test system (2010 Mac Pro w. Quadro 4000 noted.

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.

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.

Todo: Expand section on device memory types

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 `gpuMain(...)` in `device.cu`. Constant memory variables have a global scope, and are prefixed by "`devC_`" in the source code.

3.2 Performance

Todo: insert graph of performance vs. np and performance vs. Δt

.

3.2.1 Particles and computational time

3.3 Compilation

An important note is that the `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 `src/` subfolder and type `make`. The GNU Make will return the parameters passed to the individual CUDA and GNU compilers (`nvcc` and `gcc`). The resulting binary file (`sphere`) is placed in the `SPHERE` root folder. "`src/Makefile`" will also compile the raytracer.

4 Python API: Model setup

In MATLAB, enter the `mfiles/`-directory as the current folder (example: `>> cd /code/sphere/mfiles/`). For each new experiment setup, it might be a good approach to copy the software root-folder, thus saving past configurations and data.

4.1 Creating the particles

To begin with, the model surroundings and the particles are created. Particle sizes are randomly drawn from either a *log-normal-* or *uniform* distribution. These particles are situated in a fixed grid-like setup.

Using the MATLAB Editor or another editor, open the file `mfiles/initsetup.m`. Here you set:

- The filename of the output file (`fn`)
- The number of particles (`p.np`)

- The particle size distribution (`logn`: log-normal or `uni`: uniform), including
  - Mean size (`p.m`)
  - Variance (`p.v`)
- Physical parameters (`params.g`: gravity, `p.rho`: density, `p.k_n`: normal stiffness, `p.k_s`: shear stiffness), etc...)
- Time parameters (`time.dt`: computational time step length, `time.current`: starting time, `time.total`: total simulation time, `time.file_dt`: interval between output generation)

After any modifications the file needs to be saved. Write the specified model setup using the following command in MATLAB:

```
>> initsetup
```

This will write the binary file for SPHERE to read, display a histogram of the particle size distribution, and show a figure of the model setup.

4.2 Introducing heterogenieties

Until now, all grains in the model have the same physical properties. Heterogenieties can be introduced in the same way as the coloring above. The `find()`-function in MATLAB can be used to set different properties on particles, that fulfill a specific requirement, e.g. are contained between geometric border-values in the coordinate system in the initial setup.

Please note, that these deviant values need to be set *after* the general physical values in `template.m`. The following is an example, that assigns new values for cohesion (C) and angle of internal friction (ϕ) for the bottom 0.02 m particles:

```
%%Heterogenieties
%Weak bottom layer
y = p.x(:,2);
I = find(y < 0.02); %Particles defined as quantity I
phi = 1; %Angle of internal friction
DEM.particles.friction(I) = tan(phi*pi/180)*ones(length(↩
    I),1);
C = 10; %Cohesion
DEM.particles.cohesion(I) = C*ones(length(I),1);
DEM.particles.matflag(I) = 2;
```

Here, the affected particles are given a different `matflag`-value, for visual identification of the particles with different properties.

4.3 Periodic boundary conditions

For simulating mechanisms that involve large horizontal strains, as in a ring-shear apparatus, it is a good approach to make the left and right borders cyclic. That means that a particle disappearing out of the right side will reappear in

the same vertical position in the left side. Particles can affect each other with contact forces in the same way. This has the advantage that the total number of particles can be kept low in relation to a very wide model setup.

The cyclic borders are in MATLAB enabled by the command `DEM.rm.periflag = 1`. In this configuration, only the top and bottom walls are needed. A specific normal-pressure is applied in pascal ($Pa^{-1} = \frac{N}{m}$) (denote the value with a negative sign for downward oriented pressure). The shearing motion is applied by fixing the bottom particles to a horizontal velocity of zero (0), and moving the uppermost particles at a constant horizontal velocity (e.g. $0.1\frac{m}{s}$). The particles are not fixed in the vertical directions, since particles still need to be able to pass each other by volumetric expansion.

5 DEM simulation using SPHERE

After the initial model preferences have been set up in MATLAB and the binary input file is written, the DEM-calculations can be initialized from the terminal. Specify the input file name as a parameter for `SPHERE`:

```
$ ./sphere simulation_name
```

In MATLAB, the state of the calculations can be checked with:

```
>> status('simulation_name')
```

The basics of the DEM algorithm, used in `SPHERE`, is described in section **??**, page **??**. While `SPHERE` is running, output binary files are placed in the `output/` folder with the format `simulation_name.output#.bin`. On UNIX systems, the CPU usage of active processes can be monitored with the command '`top`'. GPU usage is monitored using e.g. the NVIDIA Compute Visual Profiler.

6 Python API: Data analysis

A number of preconfigured visualization methods are featured in `show.m`. It shows a figure of the particle assemblage, created on base of the data from a specific output-file. This output file is selected with the command `show('file',#)`, where the second parameter is the number of the output file (the numbering starts from zero (0)). Even though the SDEM calculations (`./start`-script) have not been completed, the latest output file can be visualized in MATLAB, as long as at least one output-file has been generated:

```
>> show('file','latest')
```

A series of graphs, including total kinetic energy, dissipation energy, number of particles and time steps as functions of time can be displayed with:

```
>> show('energy')
```

When defining a wall-position as a function of pressure, not velocity, all wall velocities can be displayed with:

```
>> show('wallspeed')
```

And if the wall-position is a function of velocity, not pressure, the wall pressures can be displayed with:

```
>> show('wallforce')
```

The total accumulated horizontal displacement of the particles:

```
>> show('sheardisp')
```

If simulating a shearing motion, it may be desired to create a graph, that shows the horizontal velocity of each particle. This is done by using the `'veloprofile'`-option in `show()`, for example:

```
>> show('file','latest','veloprofile')
```

Additional arguments can be used in the `show`-command, to visualize various parameters, i.e. pressure or strain for each particle. The `colorbar`-function adds a bar explaining the values and colors:

```
>> show('file','latest','field','pressure')
>> colorbar
>> show('file','latest','field','eplast')
>> colorbar
```

For displaying the colorbar with a fixed span:

```
>> show('file','latest','field','eplast','clim',[0;5])
>> colorbar
```

6.1 Exporting plots

The MATLAB function `exportplots(mode)` is included for automatically plotting a range of output files, and exporting them as image files. It plots the raw DEM-data from the last run experiment with show() and saves the output as image files. Open `mfiles/exportplots.m`, and check the folder, filename and graphics format settings. After saving, call the function in MATLAB:

```
>> exportplots()
```

This creates an image for each output file in the `output/` directory, and with the default settings, saves them in the folder `flic/`. During the process, MATLAB will display a waitbar, showing the progress. If needed, the export may be halted by clicking the 'Cancel' button in the waitbar window. Do not press `Ctrl-C`; after killing the export this way, a restart of MATLAB is needed before repeating the command. If, for some reason, the code crashes wile running, the waitbar is forcibly closed by:

```
>> set(0,'ShowHiddenHandles','on')
>> delete(get(0,'Children'))
```

To export the plots with particles colored according to the elastic plasticity or pressure:

```
>> exportplots('eplast')
>> exportplots('pressure')
```

To create a plot over the velocity profile for each output file, and exporting it:

```
>> exportplots('veloprofile')
```

To export all normal-, eplast-, pressure and veloprofiles as images use the command `exportall()`, which creates a series of images (combi) containing the four view modes simultaneously:

```
>> exportall(1)
```

When the goal of the export is to create an animation in the `avi` or `mpeg`-format, I recommend exporting to the `jpeg`-format, which can be chosen in `exportplots.m`. In most video editing software the `png`-format is only poorly, if at all, supported.

6.2 Importing data

All parameters and values from the simulation results can be imported into the MATLAB workspace. In MATLAB, make sure the current folder is the `mfiles/` directory inside the `DISC` folder. The DEM structure from the current configuration is loaded by `DEM = DEMload()`, which reads `input/DEM.mat`. The resulting variable structure imported into the MATLAB workspace, and consists of a hierarchy of variables, as mapped out in appendix C. While `DEMload()` loads the initial model setup, the MATLAB function `DEMloadfnr(fnr)` loads particle configurations from output file with number `fnr`.

The following is a number of examples, showing possible ways to visualize particle speed[5]. The following series of commands loads all particle positions and creates a three-dimensional plot with speed on the z-axis from output file #25:

```
>> [p, grid, time, params] = freadbin('../output/', 5);
>> x=p.x(:,1);
>> y=p.x(:,2);
>> z=sqrt(p.vel(:,1).^2+p.vel(:,2).^2);
>> plot3(x,y,z,'.')
```

Or as a triangular surface for better visualization; lone points distributed in a 3D space, visualized in 2D, are difficult to comprehend visually:

```
>> tri=delaunay(x,y);
>> trisurf(tri,x,y,z)
```

[5] Inspired by: http://geology.wlu.edu/connors/primers/Surfaces_and_Grids_in_Matlab/Surfaces_and_Grids_in_Matlab.htm

Or as a (smoothed) surface with interpolated data point values in between particles:

```
>> res=100; %Resolution in x- and y-direction
>> rangeX=min(x):(max(x)-min(x))/res:max(x);
>> rangeY=min(y):(max(y)-min(y))/res:max(y);
>> [X,Y]=meshgrid(rangeX,rangeY);
>> Z=griddata(x,y,z,X,Y);
>> surf(X,Y,Z)
```

To make a contour plot of the gridded data:

```
>> resZ=100; %Resolution in z-direction
>> rangeZ=min(z):(max(z)-min(z))/resZ:max(z);
>> contourf(X,Y,Z,rangeZ);
```

Combining contours and surfaces:

```
>> hold on
>> surf(X,Y,Z,'EdgeColor','none','FaceColor','interp','↩
    FaceLighting','phong')
>> contour3(X,Y,Z,rangeZ,'k')
>> figure(gcf)
```

Another example: Arrow field of particle velocities:

```
>> s=0.02; %Velocity scaling factor
>> quiver(p.x(:,1), p.x(:,2) ,...
   p.vel(:,1)*s, p.vel(:,2)*s,'AutoScale','off');
>> axis([0 0.2 0 0.11])
```

Appendix

A Folder structure

SPHERE sub directory and file description:

- 'cudaprofiler/': Folder for saving CUDA Compute Visual Profiler project data.
- 'doc/': Folder containing documentation
  - 'graphics/': Documentation image files
  - 'sphere-doc.pdf': This document
  - 'sphere-doc.tex': LaTeX source code for this document
  - Auxillary LaTeX files
- 'flic/': Folder for image creation plugins (not included)
  - 'eps/': EPS image format directory
  - 'gif/': GIF image format directory
  - 'png/': PNG image format directory
- 'include/': Directory for preprocessor-files, utilized in sphere.cu.
  - 'nrutil.h': Numerical recipes header file.
  - 'random.c': Contains the function ran0(...) that returns a random number.
- 'input/': Folder for simulation configuration binaries.
- 'mfiles/': Files which contain functions available to MATLAB.
  - 'bubbleplot.m': bubbleplot(p) plots particles in structure p using bubbleplot3.m
  - 'bubbleplot3.m': bubbleplot(x,y,z,r) Produces a three-dimensional bubble plot of spheres with coordinates (x,y,z) and radius (r). Created by Peter Bodin, obtained through Mathworks File Exchange[6].
  - 'freadbin.m': [p, grid, time, params] = freadbin(path, fn) reads binary file named fn in directory path, and returns MATLAB structures (appendix C).
  - 'fwritebin.m': fwritebin(path, fn, p, grid, time, params) writes MATLAB structures (appendix C) to binary file named fn in directory path.
  - 'initsetup.m': initsetup() creates particles in grid-like arrangement. The number of particles and the PSD is specified. Calling the function creates a .bin-file in the input/-folder.
  - 'status.m': status('simulation_name') writes the status of the current model run to the command window.
- 'output/': Folder containing output .bin-files. For each output time-step (time.file_dt), a new simulation_name.output#.bin is created, where # is the output time step number. Simultaneously, the status-file is updated.
  - 'simulation_name.logfile.txt': Text file containing eventual error messages from the computation.
  - 'simulation_name.output#.bin': Main SPHERE output data.
  - 'simulation_name.status.dat': File showing the completed calculation progress in percentage, and time steps.
- 'src/': Folder containing SPHERE source code.

[6] http://www.mathworks.com/matlabcentral/fileexchange/8231-bubbleplot3

B SPHERE source code variables

Variables without a prefix are located in the host memory. Variables with the prefix 'dev_' are stored in the read & write device memory, 'tex_' means data is stored in the read-only texture memory on the device.

Name	Type	Description
x[i]{.x,.y,.z} dev_x[i]{.x,.y,.z} tex_x[i]{.x,.y,.z}	float4	Particle coordinates. Returns the coordinates of particle i, optionally only the value in dimension x, y or z. E.g. x[124].x returns x-axis placement of particle 124, read from host memory.

Table 1: Some important variables in sphere.cu. The integer i refers to the particle number. Curly brackets ({}) indicate optional, exclusive extensions.

C Parameter structure in MATLAB

The DEM structure from the current configuration is loaded into MATLAB by the command `[p, grid, time, params] = freadbin(path, fn)`, which reads the specified binary file. The resulting variable structures imported into the MATLAB workspace are the following:

`p.`	`np`	Number of particles
	`psd`	Particle size distribution
	`m`	Mean particle radius
	`v`	Particle radius variance
	`radius`	Radius of all particles
	`x`	Particle positions in E^3 (x,y,z)
	`vel`	Particle linear velocities
	`angvel`	Particle angular velocities
	`vel0`	Fixed linear particle velocitis
	`force`	Sum of forces upon each particle
	`torque`	Sum of torque upon each particle
	`rho`	Particle material densities
	`k_n`	Particle normal stiffnesses
	`k_s`	Particle shear stiffnesses
	`mu`	Inter-particle contact friction coefficients
	`C`	Inter-particle tensile strengths (cohesion)
	`E`	Young's modulus for each particle
	`K`	Bulk modulus for each particle
	`nu`	Poisson's ratio for each particle
`time.`	`dt`	Computational time step length
	`current`	Current time
	`total`	Total simulation time
	`file_dt`	Interval between output file generation
	`step_count`	Current step count
`grid.`	`nd`	Number of dimensions
	`origo`	Cartesian space origo
	`num`	Number of grid cells in each dimension
	`L`	Grid length in each dimension
`params.`	`g`	Gravitational acceleration
	`global`	Equals 1: Particle mechanical parameters are global, equals 0: Particle mechanical parameters are individual and set in the `p` structure.

Table 2: MATLAB variable structure

The values are written using the corresponding command `fwritebin(path, fn, p, grid, time, params)`.

D Sample: initsetup.m

```
function initsetup()
% initsetup() creates a model setup of particles in 3D with cubic
% packing. Specify PSD and desired number of particles, and the
% function will determine the model dimensions, and fill the space
% with a number of particles from a specified particle size distribution.
close all;

%Simulation project name
simulation_name = '3dtest';

%Physical, constant parameters
params.g = 9.80665; %standard gravity, by definition 9.80665 m/s^2

% No. of dimensions
grid.nd = 3;
grid.origo = [0 0 0]; %Coordinate system origo

% Number of particles
p.np = 5e2;

% Create grid
grid.num = ceil(nthroot(p.np,grid.nd)) * ones(grid.nd, 1); %Square/cubic
p.np = grid.num(1)^grid.nd;

%% Particle size distribution
p.psd = 'logn'; %PSD: logn or uni

p.m = 440e-6; %Mean size
p.v = p.m*0.00001; %Variance
%p.m = 22; %Mean size
%p.v = p.m*0.001; %Variance

p.radius = zeros(1, p.np);
p.x      = zeros(p.np, grid.nd);

if strcmp(p.psd,'logn') %Log-normal PSD. Note: Keep variance small.
    mu = log((p.m^2)/sqrt(p.v+p.m^2));
    sigma = sqrt(log(p.v/(p.m^2)+1));
    p.radius = lognrnd(mu,sigma,1,p.np); %Array of particle radii
elseif strcmp(p.psd,'uni') %Uniform PSD between rmin and rmax
    rmin = p.m - p.v*1e5;
    rmax = p.m + p.v*1e5;
    %rmin = 0.1*dd; rmax = 0.4*dd;
    p.radius = (rmax-rmin)*rand(p.np,1)+rmin; %Array of particle radii
end

%%Display PSD
figure(2);hist(p.radius);
title(['PSD: ' num2str(p.np) ' particles, m = ' num2str(p.m) ' m']);


%% Other particle parameters
p.vel    = zeros(p.np, grid.nd); % Velocity vector
```

```
54 p.angvel = zeros(p.np, grid.nd); % Angular velocity
55 p.vel0   = zeros(p.np, grid.nd); % Pre-velocity vector
56 p.force  = zeros(p.np, grid.nd); % Force vector
57 p.torque = zeros(p.np, grid.nd); % Torque vector
58
59 params.global = 1; % 1: Physical parameters global, 0: individual per particle
60 p.rho    = 3600*ones(p.np,1);    % Density
61 p.k_n    = 109680*ones(p.np,1);  % Normal stiffness
62 p.k_s    = 109680*ones(p.np,1);  % Shear stifness
63 p.mu     = 1*ones(p.np,1);       % Inter-particle contact friction coefficient
64 p.C      = 0*zeros(p.np,1);      % Inter-particle cohesion
65 p.E      = 10e9*ones(p.np,1);    % Young's modulus
66 p.K      = 38e9*ones(p.np,1);    % Bulk modulus
67 p.nu     = 0.38*ones(p.np,1);    % Poisson's ratio
68
69 %Time parameters
70 time.dt         = 1e3*0.5*min(sqrt((p.rho(:).*p.radius(:).^2)./p.K(:))); %Computational delta t
71 time.current    = 0.0;
72 time.total      = 1.0; %Total simulation time [s]
73 time.file_dt    = 0.1; %Interval between output#.bin generation [s]
74 time.step_count = 0;
75
76 %% Calculate particle coordinates
77 %Grid unit length. Maximum particle diameter determines grid size
78 GU = 2*max(p.radius)*1.10; % Ten percent margin
79 %Model world side length (cubic arrangement)
80 grid.L = [GU*grid.num(1) GU*grid.num(2) GU*grid.num(3)];
81
82 %% Particle coordinates by filling grid.
83 x = GU/2:GU:grid.L(1);
84 y = GU/2:GU:grid.L(2);
85 z = GU/2:GU:grid.L(3);
86
87 % Wall coordinates
88  walls = [0,0,0          , grid.L(1),0,0          , grid.L(1),0,grid.L(3)          , 0,0,grid.L(3)          ; ... %Bottom wall (x,z plane)
89           0,grid.L(2),0  , grid.L(1),grid.L(2),0  , grid.L(1),grid.L(2),grid.L(3) , 0,grid.L(2),grid.L(3) ; ... %Top wall (x,z plane)
90           0,0,0          , 0,grid.L(2),0          , grid.L(1),grid.L(2),0          , grid.L(1),0,0          ; ... %Front wall (x,y plane)
91           0,0,grid.L(3)  , 0,grid.L(2),grid.L(3)  , grid.L(1),grid.L(2),grid.L(3) , grid.L(1),0,grid.L(3) ; ... %Back wall (x,y plane)
92           0,0,0          , 0,grid.L(2),0          , 0,grid.L(2),grid.L(3)          , 0,0,grid.L(3)          ; ... %Left wall (y,z plane)
93           grid.L(1),0,0  , grid.L(1),grid.L(2),0  , grid.L(1),grid.L(2),grid.L(3)]; %Right wall (y,z plane)
94
95 %% Plot
96 figure(1);
97
98 % %draw walls
99     for i=1:6,
```

```
100         line(walls(i,[1,4]),walls(i,[2,5]),walls(i,[3,6]),'↩
                linewidth',2,'color','k');
101         line(walls(i,[7,10]),walls(i,[8,11]),walls(i,[9,12]),'↩
                linewidth',2,'color','k');
102         line(walls(i,[4,7]),walls(i,[5,8]),walls(i,[6,9]),'↩
                linewidth',2,'color','k');
103         line(walls(i,[10,1]),walls(i,[11,2]),walls(i,[12,3]),'↩
                linewidth',2,'color','k');
104     end;
105
106 [X Y Z] = meshgrid(x,y,z);
107 X=X(:); Y=Y(:); Z=Z(:);
108
109 p.x = [X Y Z];
110
111 %Particle positions randomly modified by +/- 5 percent
112 p.x = p.x .* (rand(p.np, grid.nd)*0.1 + 0.95);
113
114 %% Plot particles in bubble plot
115 bubbleplot(p);
116
117 %% Write output binary
118 fwritebin('../input/', [simulation_name '.bin'], p, grid, time, ↩
        params);
119
120 end
```

— End of file —