SPH-DEM Documentation

Martin Robinson

Contents

1 Introduction

SPH-DEM is primarily an implementation of the SPH method, described in more detail in the following papers [1, 4, 2]. It has the following main features

- Written in C++
- Modular structure separates the parallel data structure from the SPH/DEM equation. Uses C++ templates to efficiently apply arbitrary functions on particles and their neighbours
- Fully parallel implementation using MPI
- Output files use VTK format
- SPH Functionality
  - Weakly compressible formulation, as described in [4]
  - Lenard-Jones type boundary particles, periodic boundaries and ghost boundaries
  - Variety of different viscosity terms and kernel options
  - First order corrected kernel derivatives
- DEM Functionality
  - Linear Spring Dashpot and Lubrication contact models
  - SPH boundary particles can be used for walls
- SPH-DEM coupling based on Locally Averaged Navier-Stokes Eqns. Drag terms: De Felice, Ergun and Stokes. Details of this implementation can be found in [3]

2 Basic Usage

Each project/simulation has its own directory under the root source directory for SPH-DEM. For example, to create a new project called *testProject*:

```
cd $SPH_DEM_SOURCE_DIR
mkdir testProject
cd test Project
```

It is helpful to simply copy the contents of an older project in order to initialise the new project. For example, using the taylor green vortex test case *taylorGreen*:

```
cp taylorGreen/* testProject/
```

Then navigate to the new project, edit the configuration files (not shown, see Section 3 for details), and then compile the project

```
cd testProject
...edit configuration files...
make
```

You can setup the initial state of the project using the *setup* executable, and then run the simulation using *run*.

```
./setup INITDATA_NAME
./run INITDATA_NAME START_TIMESTEP OUTPUTDATA_NAME
```

Here INITDATA_NAME is the name given to the initial data files, START_TIMESTEP is the timestep to start the simulation (normally 0 if you are not restarting a simulation) and OUTPUTDATA_NAME is the name given to the output data files. See Section 5 for more details of the input/output data formats.

3 Project Description

This sections describes the per-project configuration and setup. The main configuration options can be set in *customConstants.h* and *setup.cpp*. The header file *customConstants.h* contains a list of c++ constants and #defines that can customise each simulation. The source file *setup.cpp* is compiled to an executable that constructs and writes out the initial data for the simulation.

Each project also has two classes *customSim* and *customOutput* that allow the user to customise the simulation.

3.1 Custom Constants

Different options in SPH-DEM are controlled by #define statements and constants defined in the *customConstants.h* file in each project subdirectory. For example, here is a listing for the included *taylorGreen* project.

```
#ifndef CUSTOMCONSTANTS_H
#define CUSTOMCONSTANTS_H

#include <cmath>

#define _2D_
#define NDIM 2
#define INCOMPRESS

const double PI = 3.14159265;

const double RMIN[NDIM] = {0,0};
const double RMAX[NDIM] = {1.0,1.0};

#define TIME_OVERIDE 0
#define START_TIME 0
#define NSTEP_OVERIDE 0
#define START_NSTEP 0

const double GAMMA = 1.4;
const double HFAC = 1.5;

const int PERIODIC[NDIM] = {1,1};

const double DENS = 1000.0;
const double REFD = 1000.0;
const int GHOST[2*NDIM] = {0,0,0,0};
const double DENS_DROP[NDIM] = {0,0};

#define WENDLAND
#define VISC_MONAGHAN
#define VORT_LEASTSQUARES
```

```

#define NO_ANTICLUMPING
#define CONST_H

const double MAXTIME = 6;
const double DAMPTIME = 0;
const int OUTSTEP = 100;
const int RESTART_EVERY_N_STEPS = 100;
const int REINIT_DENS_EVERY_N_STEPS = 2000000;
const int REINIT_DENS_AT_N_STEPS = 2000000;
const int DAMP_STEPS = 0;

const double NVORT = 2;
const double REYNOLDS_NUMBER = 100;
const double VREF = 1.0;
const double VMAX = VREF;
const double VISCOSITY = VREF*(RMAX[0]-RMIN[0])/REYNOLDS_NUMBER;

const double CSFAC = 10.0;
const double SPSOUND = CSFAC*VMAX;
const double PRB = pow(REFD/DENS,6)*pow(SPSOUND,2)*REFD/7.0;

const int NX = 20;
const int NY = 20;
const double PSEP = (RMAX[0]-RMIN[0])/NX;
const double H = HFAC*(RMAX[0]-RMIN[0])/NX;
const double BFAC = 1.0;

const int MAX_NUM_PARTICLES_PER_CPU = NX*NY+1;
const double GRIDSEP = PSEP;
```

This file is a mix of define statements and constants that are used for the main SPH-DEM program, and others (e.g. NVORT, REYNOLDS_NUMBER) that are only used within the *taylorGreen* project directory.

The rest of the section is devoted to listing all the available defines and constants used by the SPH-DEM program. The comments describe each option and the code gives an example of use.

3.1.1 Simulation Geometry

```
/*
 * number of spatial dimensions. Can be 1, 2 or 3.
 */
#define NDIM 2
/*
 * RMIN and RMAX are arrays containing the minimum and maximum
      extents
 * of the simulation
 */
const double RMIN[NDIM] = {0,0};
const double RMAX[NDIM] = {1.0,1.0};

/*
 * The intial (and average) particle spacing between sph
      particles
 */
const double PSEP = (RMAX[0]-RMIN[0])/NX;

/*
 * spacing between boundary particles
 */
```

```
const double BFAC = 1.0;

/*
 * The spacing of the regular grid (if used)
 */
const double GRIDSEP = PSEP;
```

3.1.2 Boundary Conditions

```
/*
 * Each entry in PERIODIC (0 or 1) tells the program if the
     simulation
 * domain is periodic in that spatial dimension
 */
const int PERIODIC[NDIM] = {1,1};

/*
 * Each entry in GHOST (0 or 1) tells the program if that
     boundary is a ghost or
 * mirror boundary. Entries 0 and 1 refer to the lower and upper
      boundaries
 * in the x direction, 2 and 3 refer to the lower and upper
     boundaries in the y
 * direction, and so on.
 */

const int GHOST[2*NDIM] = {0,0,0,0};

/*
 * Each entry in DENS_DROP gives the relative density drop
     experianced by the
 * SPH particles as they cross the periodic boundary in that
     spatial dimension
 */
 const double DENS_DROP[NDIM] = {10,0};
```

3.1.3 Timing

```

/*
 * Time at which simulation ends
 */
const double MAXTIME = 6;

/*
 * Time at which damping ends (0 for no damping)
 */
const double DAMPTIME = 0;

/*
 * Number of output timesteps (time between output timesteps is
     therefore
 * constant during the simulation
 */
const int OUTSTEP = 100;

/*
 * After RESTART_EVERY_N_STEPS timesteps, a restart file is
     written out
 * so that the simulation can be restarted at this point
 */
const int RESTART_EVERY_N_STEPS = 100;
```

```

/*
 * TIME_OVERIDE (0 or 1) tells the program to override the
      starting time read in
 * from the datafiles
 *
 * START_TIME is the time used if TIME_OVERRIDE = 1
 */
#define TIME_OVERIDE 0
#define START_TIME 0

/*
 * NSTEP_OVERIDE (0 or 1) tells the program to override the
      starting timestep
 * number read in from the datafiles
 *
 * START_NSTEP is the timestep number used if NSTEP_OVERRIDE = 1
 */
#define NSTEP_OVERIDE 0
#define START_NSTEP 0
```

3.1.4 SPH options

```
/*
 * GAMMA is the exponent used in the SPH pressure equation of
      state
 */
const double GAMMA = 1.4;

/*
 * REFD is the reference density used in the pressure equation
      of state
 */
const double REFD = 1000.0;

/*
 * The speed of sound. This is artificialy set for
      incompressible simulation
 * to 10 times the maximum velocity.
 */
const double SPSOUND = CSFAC*VMAX;

/*
 * PRB is the multiplicative constant in the pressure equation
      of state.
 */
const double PRB = pow(REFD/DENS,6)*pow(SPSOUND,2)*REFD/7.0;

/*
 * Each SPH smoothing length is HFAC times the average particle
      spacing PSEP
 */
const double HFAC = 1.5;

/*
 * Options to re-initialise the sph particle density ever n
      steps or once
 * at step n. If REINIT_DENS_MLS is set use a Moving Least
      Squares kernel
 * to reinit the density. Otherwise use a normal sph density sum
 */
const int REINIT_DENS_EVERY_N_STEPS = 2000000;
const int REINIT_DENS_AT_N_STEPS = 2000000;
#define RINIT_DENS_MLS
```

```

/*
 * Different sph kernels. If none are chosen the standard cubic
     spline is used
 */
#define WENDLAND
#define QUINTIC
#define HANN

/*
 * Different viscosity terms. At least one must be chosen. The
     ARTIFICIAL
 * viscosity can be added to any of the others.
 */
#define VISC_MONAGHAN
#define VISC_MORRIS
#define VISC_CLEARY
#define VISC_ARTIFICIAL
const double ALPHA_ARTIFICIAL = 0.1;

/*
 * kinematic vicosity of the fluid
 */
const double VISCOSITY = VREF*(RMAX[0]-RMIN[0])/REYNOLDS_NUMBER;

/*
 * Do not use the monaghan anti-clumping term (this is used by
     default)
 */
#define NO_ANTICLUMPING

/*
 * Do not vary the particle smoothing length h with density
 */
#define CONST_H

/*
 * Use the variable h correction terms proposed in
 * Price, D., 2012. Smoothed particle hydrodynamics and
     magnetohydrody-
 * namics. Journal of Computational Physics 231, 759  794.
 */
#define VAR_H_CORRECTION

/*
 * Use the corrected kernel gradient terms proposed in
 * J. Bonet, T.-S.L. Lok, Variational and momentum preservation
     aspects of
 * Smooth Particle Hydrodynamic formulations, Computer Methods
     in Applied
 * Mechanics and Engineering, Volume 180, Issues 1  2, 15
     November 1999, Pages
 * 97-115
 */
#define CORRECTED_GRADIENT
```

3.2 Setup

Below is shown an example of the code neccessary to write out initialisation data for SPH-DEM. Note that this is not a neccessary step to running a simulation. If the initialisation data can be obtained in another fashion, or from a restart file from a previously run simulation, then this can be used to start another SPH-DEM simulation.

```
int main(int argc, char *argv[]) {
   if (argc != 2) {
      cout << "Usage: setup outfilename" << endl;
      return(-1);
   }
   string filename = argv[1];

   vector<Cparticle> ps;
   Cparticle p;

   for (int i=0;i<NX;i++) {
      for (int j=0;j<NY;j++) {
         p.tag = ps.size()+1;
         p.r = (i+0.5)*PSEP+RMIN[0],(j+0.5)*PSEP+RMIN[1];
         p.dens = DENS;
         p.mass = PSEP*PSEP*DENS;
         p.h = H;
         p.v = -VREF*cos(2.0*PI*p.r[0])*sin(2.0*PI*p.r[1]),VREF*
              sin(2.0*PI*p.r[0])*cos(2.0*PI*p.r[1]);
         p.vhat = p.v;
         p.iam = sph;
         ps.push_back(p);
      }
   }

   vector<vector<double> > vprocDomain;
   vector<Array<int,NDIM> > vprocNeighbrs;
   vector<particleContainer > vps;
   vectInt split;
   split = 1,1;
   Nmisc::splitDomain(ps,split,vps,vprocDomain,vprocNeighbrs);

   CglobalVars globals;
   Cio_data_vtk ioFile(filename.c_str(),&globals);

   int nProc = product(split);
   for (int i=0;i<nProc;i++) {
      globals.mpiRank = i;
      for (int j=0;j<NDIM*2;j++)
         globals.procDomain[j] = vprocDomain[i][j];
      globals.procNeighbrs = vprocNeighbrs[i];
      ioFile.setFilename(filename.c_str(),&globals);
      ioFile.writeGlobals(0,&globals);
      ioFile.writeRestart(0,vps[i],&globals);
      ioFile.writeDomain(0,&globals);
   }
}
```

The above code example can be split into three sections. In the first the sph particles are created and added to the vector of particles. In the second the particles and domain are spit evenly among the number of CPUs chosen by the user. In the third the result is written to the data files

3.2.1 Creating new SPH particles

```
   vector<Cparticle> ps;
   Cparticle p;

   for (int i=0;i<NX;i++) {
      for (int j=0;j<NY;j++) {
         p.tag = ps.size()+1;
         p.r = (i+0.5)*PSEP+RMIN[0],(j+0.5)*PSEP+RMIN[1];
```

```
            p.dens = DENS;
            p.mass = PSEP*PSEP*DENS;
            p.h = H;
            p.v = -VREF*cos(2.0*PI*p.r[0])*sin(2.0*PI*p.r[1]),VREF*
                sin(2.0*PI*p.r[0])*cos(2.0*PI*p.r[1]);
            p.vhat = p.v;
            p.iam = sph;
            ps.push_back(p);
        }
    }
```

The class *Cparticle* contains all the data for a single sph particle. Here we are creating a 2D regular grid of particles that completly fill the simulation domain. So we loop through the number of particles chosen (using NX and NY) and create the particles. We are simulating a taylor green vortex, so the initial velocity of the particles is set using the analytical expression for the vortex velocity field.

Each particle has a volume equal to a square of side PSEP, which along with the density DENS gives the particle mass. The smoothing length is given by H (which in turn is given by HFAC*PSEP) and the particle type is set to "sph". The two velocities, *v* and *vhat*, are set equal. The former is the "true" velocity field (used to calculate viscosity) while the latter is an (optionally) smoothed velocity which is used to update the particle positions (and also used to calculate the change in density with time).

3.2.2 Split data among CPUs

```

    vector<vector<double> > vprocDomain;
    vector<Array<int,NDIM> > vprocNeighbrs;
    vector<particleContainer > vps;
    vectInt split;
    split = 1,1;
    particleContainer ps;
    Nmisc::splitDomain(ps,split,vps,vprocDomain,vprocNeighbrs);
```

The *splitDomain* procedure splits the particles among equal sized subdomains according to the *split* vector given by the user. This example only uses one CPU. If two CPUs were required, then this could be modified to

```
    split = 2,1;
```

The vectors *vprocDomain* and *vprocNeighbrs* contain the side min/max point information of the domains and neighbourhood information (i.e. which domains are next to each other). The resultant vector of vector of particles is given in *vps*.

3.2.3 Write Data

```
    CglobalVars globals;
    Cio_data_vtk ioFile(filename.c_str(),&globals);

    int nProc = product(split);
    for (int i=0;i<nProc;i++) {
        globals.mpiRank = i;
        for (int j=0;j<NDIM*2;j++)
            globals.procDomain[j] = vprocDomain[i][j];
        globals.procNeighbrs = vprocNeighbrs[i];
        ioFile.setFilename(filename.c_str(),&globals);
        ioFile.writeGlobals(0,&globals);
        ioFile.writeRestart(0,vps[i],&globals);
        ioFile.writeDomain(0,&globals);
    }
```

This code segment writes out the particles, domain information and global variables to restart files, which can then be used to initialise the simulation. Note that this is the same procedure used when writing periodic restart information while the simulation is running, so the initialisation and restart procedure for SPH-DEM are identical.

3.3 Custom Simulation and Outputs

Within each project directory are defined two classes, *customSim* and *customOutput*. These allow the user to add arbitrary code into the simulation to, for example, move boundary particles, insert/remove particles or apply a custom postprocessing algorithm. They are both derived from base classes which do nothing, so if no modification to the base simulation is neccessary the user only has to leave the classes empty.

The base class for *customSim* is as follows:

```
class CcustomSimBase {
    public:
        CcustomSimBase(CdataLL *_data,double _time): data(_data),
            time(_time) {}
        double getTime() { return time; }
        virtual void beforeStart(double newTime) { time = newTime;
            }
        virtual void beforeMiddle(double newTime) { time = newTime
            ; }
        virtual void beforeEnd(double newTime) { time = newTime; }
        virtual void afterEnd(double newTime) { time = newTime; }
        virtual vect vFilter(const vect vin, const vect rin) {
            return vin; }
        virtual vect rFilter(const vect rin) { return rin; }

    protected:
        double time;
        CdataLL *data;
};
```

This class defines a number of virtual functions, *beforeStart*, *beforeMiddle* etc. which are called either before or after the start, middle or end of each timestep. Users can reimplement these functions in the *customSim* class to insert code at these points. In addition, the *vFilter* and *rFilter* functions can be used to filter the position and velocity of each particles respectivly before they are writen to output files.

The protected variable *data* is a pointer to the main particle data structure, and the description for its use can be seen in Section 4.

The listing for the base class for *customOutput* is given below.

```
class CcustomOutputBase {
    public:
        CcustomOutputBase(CdataLL *_data): data(_data) {}
    virtual void calcOutput(int outstep,CcustomSim *custSim,
        Cio_data_vtk *io) {}
  protected:
    CdataLL *data;
};
```

This contains one function, *calcOutput* which is called just before the files are written every output timestep. Users can therefore apply any custom post-processing here.

4 Writing New Functions

In order to add custom simulation and/or post-processing code to the simulation the user will need to use the included particle data structure *CdataLL*, described in *dataLL.h*, *dataLL.cpp* and *dataLL.impl.h* in the main source code directory. This class holds the particle data and handles the domain geometry, parallel processing and the application of custom operations to the particle data. It is the latter functionality we will use in this section.

Two of *CdataLL*'s methods are:

```
template <class T, void Thefunct(Cparticle &,CglobalVars &,T &),
     bool Iffunct(Cparticle &)>
void traverse(T &);

template <class T, void Thefunct(Cparticle &,vector<Cparticle *>
     &,CglobalVars &,T &), bool Iffunct(Cparticle &)>
void neighboursGroup(T &);
```

The first applies the function *Thefunct* to all the particles in the simulation that satisfy the function *Iffunct*. The second applies the function *Thefunct* to all the particles (that satisfy *Iffunct* and their neighbouring particles in the simulation. The template class *T* and the corresponding input variable are optional and are used to pass data to the functions.

In order to use the first method, *traverse*, we first need to define an inline function to apply to the particles.

```
inline void addGravity(Cparticle &p,CglobalVars &g) {
   p.ff[2] -= 9.81;
   p.f[2] -= 9.81;
}
```

This is a simple function that adds gravity to a single particle. Too apply this function to all the particles in the simulation we use *data*, which is a pointer to the main *CdataLL* datastructure, to call the *traverse* method. We also use the *ifSph* function to only apply *addGravity* to the sph particles in the simulation.

```
data->traverse<addGravity,Nsph::ifSph>();
```

Looking at the *customSim* and *customOutput*, you can see that the *data* pointer is a member variable in each class. As described previously, this points to the main particle data structure of the simulation.

To use the *neighboursGroup* method, we once again define an inline function to apply to the particles. But this time the definition of the function is slightly different.

```
inline void calcDensity(Cparticle &p, vector<Cparticle *> &
    neighbrs, CglobalVars &g) {
   p.dens = p.mass*W(0,p.h);
   for (int i=0;i<neighbrs.size();i++) {
      Cparticle *pn = neighbrs[i];
      if ((pn->iam != sph)&&(pn->iam != sphBoundary)) continue;
      vect dr = p.r-pn->r;
      double r = len(dr);
      p.dens += pn->mass*W(r/p.h,p.h);
   }
}
```

This function loops over all the neighbouring particles and calculates a standard sph density interpolation.

To use this function, we simple call the *neighboursGroup* method from the main particle datastructure. We will also use the *ifSphOrSphBoundary* function to only apply *calcDensity* to the sph or boundary sph particles.

```
data->neighboursGroup<calcDensity,ifSphOrSphBoundary>();
```

5 IO Format

The datafiles written by SPH-DEM are organised around a base filename, given by the user. As an example for the rest of this section, we will use *results* as a base filename. Using this, the list of datafiles are

resultsDDDDD_NN.vtu :	These are the particle properties (position, velocity, density etc) written in the VTK file format. DDDDD is the timestep number and NN is the CPU number. Each CPU writes its own particles to a separate file.
resultsDDDDD.pvtu :	These are small files that just point to each *resultsDDDDD_NN.vtu* file for timestep DDDDD
resultsGlobalsNN.dat :	These are csv text files, using spaces for separators instead of a comma, that contain the global variables for each timestep. Things like total kinetic energy, momentum, time, CPU time, timestep etc. Each CPU writes out a different globals datafile, but the *resultsGlobals0.dat* file is the file that is of main interest, since all the “total” variables are summed across the CPUs in this file. See the first line in the file to see the list of global variables written.
resultsDomainNN.dat :	This is a text file containing the domain extents for each CPU. This is primarily for internal use by SPH-DEM.
resultsRestartDDDDD_NN.vtu :	These are sim-ilar to *resultsDDDDD_NN.vtu*, but are used to restart the program. They therefore use better accuracy (doubles instead of floats) and contain more particle properties that the normal output datafiles.

6 SPH DEM Functionality

This section describes the DEM functionality added to SPH-DEM and the new user options for both DEM and the coupling between the sph and dem particles.

6.1 Dem Particle

A new particle type *dem* has been added. When creating this new particle, create a new particle as per the sph particle, but with the following changes:

```
...
p.dens = DEM_DENS;
p.h = DEM_SEARCH_RADIUS;
p.mass = DEM_VOL*DEM_DENS;
p.iam = dem;
...
```

To help with the initialisation of dem projects, a few helper functions are included in *misc.h*. These are:

```
/*
 * Add regular grid of dem particles within the rectangular block defined by
 * the points minRange and maxRange. The grid spacing is calculated using the
 * desired porosity
 */
void addGridDEMparticles(vector<Cparticle> &ps,const double minRange[NDIM],const double maxRange[NDIM],const double porosity);

/*
 * Add dem particles randomly within the rectangle defined by the points
 * minRange and maxRange. The number of particles inserted is calculated
 * using the desired porosity
 */
void addRandomDEMparticles(CdataLL *data,const double minRange[NDIM],const double maxRange[NDIM],const double porosity);

/*
 * Add dem particles randomly within a vertical cylinder. The origin point,
 * radius and height define the cylinder dimensions. The number of particles
 * inserted is calculated using the desired porosity
 */
void addRandomDEMparticlesCyl(CdataLL *data,const vect origin,const double radius,const double height,int n);
```

6.2 New Custom Constants

Some new constants and options are included to handle the new dem and sph-dem functionality. These are described below.

6.2.1 DEM

```
/*
 * Radius of dem particles. Used in dem-dem contact calculation and
 * dem-sph drag caculation
 */
const double DEM_RADIUS = 0.0005;

/*
 * Volume of dem particles.
 */
const double DEM_VOL = (4.0/3.0)*PI*pow(DEM_RADIUS,3);

/*
 * Density of dem particles
 */
const double DEM_DENS = DENS*1.5;

/*
 * reduced mass of dem particles. since all dem particles
 * have equal mass this will be equal to 0.5*dem_mass
 */
const double DEM_MIN_REDUCED_MASS = 0.5*DEM_MASS;

/*
 * spring and damping constants for linear spring dash-pot
 * Also used for lubrication contact force if particles are
 * in direct contact
 */
const double DEM_K = 1.0e01;
const double DEM_GAMMA = 0.0004;

/*
 * max number of dem particles. Used to allocate memory
 */
const int MAX_NUM_DEM_PARTICLES = 1.0*NDEM;

/*
 * all dem particles are fixed in space
 */
#define FIXED_DEM

/*
 * no dem-dem contacts
 */
#define NO_DEM_CONTACTS

/*
 * All dem particles translate as a solid block
 * Use to simulate a porous block
 */
#define DEM_BLOCK

/*
 * Two different contact models implemented. LINEAR is a
 * linear spring-dashpot model and LUBRICATION models
 * lubrication force between particles immersed in water.
 * ROUGHNESS_EPSILON is the separation distance, scaled
 * by particle radius, at which the lubrication force swaps
 * to a spring-dashpot model.
 */
#define LINEAR
#define LUBRICATION
const double ROUGHNESS_EPSILON = 0.01;
```

6.2.2 SPH-DEM Coupling

```
/*
 * Necessary to switch on sph-dem coupling and
 * any dem functionality
 */
#define LIQ_DEM

/*
```

```
 * defines the coupling radius used for sph-dem coupling.
 * This is used when calculating porosity and interpolating
 * particle drag from dem to sph phases.
 */
const double LIQ_DEM_COUPLING_RADIUS = 8*DEM_RADIUS;

/*
 * Normally the dem particles collide with any boundary
 * particle boundaries. This turns this off so that the user
 * can define their own custom dem boundaries (in customSim)
 */
#define LIQ_DEM_CUSTOM_WALL_CONTACTS

/*
 * Normally the drag calculated on the dem particles is
    transferred
 * to the sph particles via a kernel interpolation. This turns
    this
 * off and calculates the drag separatly on the sph particles
 */
#define LIQ_DEM_SEPARATE_DRAGS

/*
 * Turns off the drag on the sph particles
 */
#define LIQ_DEM_ONE_WAY_COUPLE

/*
 * Uses a variable timestep condition for the liq-dem coupling
 */
#define LIQ_DEM_VARIABLE_TIMESTEP
```

References

[1] R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamic: theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, 181:375–389, 1977.

[2] J. J. Monaghan. Smoothed particle hydrodynamics. *Reports of Progress in Physics*, 68:1703–1759, August 2005.

[3] M. Robinson, S. Luding, and M. Ramaioli. Fluid-particle flow modelling and validation using two-way-coupled mesoscale SPH-DEM. *submitted to Int. Journal of Multiphase Flow*, 2012.

[4] M. Robinson and J.J. Monaghan. Direct numerical simulation of decaying two-dimensional turbulence in a no-slip square box using smoothed particle hydrodynamics. *International Journal for Numerical Methods in Fluids*, 2011.