

# Recent advances in MercuryDPM

Anthony R. Thornton, Timo Plath, Igor Ostanin, Holger Götz, Jan-Willem Bisschop, Mohamed Hassan, Raïsa Roeplal, Xiuqi Wang, Sahar Pourandi and Thomas Weinhart

**Abstract.** In this paper we introduce the open-source code *MercuryDPM*: a code for simulating discrete particles. The paper discusses software and management issues that may be interesting for the developers of other open-source codes. Then we review the new features that have been added since the last publication: an improved Hertz-Mindlin model; a new liquid bridge model of Lian and Seville; a droplet-spray model; better support for re-creating complex, measured particle size distributions; a new implementation of rigid clumps; an implementation of elastic membranes; a wear model for walls; a soft-kill feature and a cloud-deployment interface for AWS.

## 1. General Introduction

*MercuryDPM* is a code for discrete particle simulations: It simulates the motion of particles, or atoms, by applying forces and torques that stem either from external body forces (e.g. gravity) or from particle interaction laws. For granular particles, these are typically contact forces (elastic, plastic, viscous, frictional) or short-range adhesive forces (liquid bridges, van der Waals forces), while for molecular simulations, forces typically stem from elastic interaction potentials (e.g. Lennard-Jones). The code has been developed extensively for granular applications, but could be adapted to include long-range interactions as well.

The code is open-source and has many developers. Thus, new features are added regularly. To keep users and developers up-to-date, we regularly publish conference proceedings detailing either the newly developed features [1, 2], discussing what is in development [3] or summarising key features [4, 5, 6]. Furthermore, we released a full paper [7] in 2020, documenting every *MercuryDPM* feature developed until that point. The aim of this conference

---

We would like to thank Donna Fitzsimmons (MercuryLab B.V.) for aiding with the creation of the figures and careful proof reading.

proceeding is to continue this sequence, and document the features, and other changes to the code, that have been developed since the full paper was written [7].

## 2. About MercuryDPM

*MercuryDPM* was started in 2009 by Anthony Thornton and Thomas Weinhart with the aim of creating a discrete particle method software able to solve complex industrial scenarios. This required several features which now form the backbone of *MercuryDPM*: (i) A flexible implementation allowing complex wall and boundary conditions, (ii) a neighbourhood detection algorithm capable of dealing with highly polydisperse particle packings [8], and (iii) an analysis tool able to extract the most relevant information from the huge amount of data generated by these simulations [9]. The code also has been coupled to the continuum solver oomph-lib [10] to simulate particle interactions with elastic solids, as well as multiscale coupling to simulate granular materials in a computationally efficient way [11].

The code has been open-source since it was started. We started with a GPL license but very quickly moved to a BSD 3-clause license, as this was felt to be more open to external development and simpler to understand. Since its conception, both its user and developer base has grown, with 48 people so far contributing significantly to the code base. Information about past and current contributors can be found on the team page of the *MercuryDPM* website, [www.mercurydpm.org](http://www.mercurydpm.org).

*MercuryDPM* is a versatile, object-oriented C++ code which (we hope) is easy to understand. It is regularly tested on several Linux distributions, Mac OS and Windows 10. To avoid breaking already existing code, a suite of over 260 self-tests have been developed, testing each feature of the code. Developing new applications in the software is straightforward: The user specifies the particulars of their simulation (initial positions, inflow, outflow, walls, interaction parameters) in a single driver file, which calls the *MercuryDPM* kernel to execute the simulations. All kernel features are documented, and there are many sample driver codes demonstrating the features.

When the code was first started, we used a self-hosted svn server. However, as the user and developer base has grown this was not maintainable; therefore, in May 2022 we moved to a git repository on bitbucket. This version of the code can be found at <https://bitbucket.org/mercurydpm/mercurydpm>. Moving to git has accelerated the return of features from developers outside the core team.

Building the code is managed using cmake, the test suite via ctest, for code maintenance (bug reporting/tracking, release planning, etc) we use the Atlassian tools: Jira and Confluence. For visualisation we use both Paraview [12] and an in-house code from Stefan Luding: XBalls, which is now distributed with *MercuryDPM*.

### 3. Release strategy and version number

*MercuryDPM* has a hybrid development pattern: Firstly, we have an open-development model, where the master branch (which includes the newest features) is openly available; plus most of the active feature branches, so code can be seen and used ahead of final testing and journal publication. We encourage new developers to use branches in the main repository so others can see their features in development, but we do not insist on this. The master branch can be accessed at <https://bitbucket.org/mercurydpm/mercurydpm/src/master>.

Secondly, we have returned to doing stable release versions, approximately once a year. After deployment of a release, it will remain in an **Alpha** state up until 28 days of the last bug fix. At the alpha stage the release notes are drafted. The following 91 days it will be in **Beta** state until it is considered stable. From version 1.0.0 onward, *MercuryDPM* will use the following convention of three numbers, *a.b.c*: The version number, *a*, is incremented when an interface changes; the second number, *b*, when new features are added; and the third number, *c*, for a bug fix to a released version. This scheme complies to semantic versioning rules and for the future, we aim to automate deployment by introduction of conventional commits. The current release can be accessed at <https://bitbucket.org/mercurydpm/mercurydpm/src/1.0.Alpha>.<sup>1</sup>

### 4. Hertz-Mindlin (improved) no-slip contact model

This section discusses changes made to the Hertz-Mindlin contact model, which was changed for 1.x series. The changes are best demonstrated by the driver codes `Hertzian2DSelfTest.cpp` and `MindlinSelfTest.cpp`.

The Hertz-Mindlin contact model uses Hertz theory to determine the normal elastic force between contacting spherical particles and a tangential force model established by Mindlin and Deresiewicz [13]. Additionally, rolling friction can be added according to the model of Luding [14]. All three models have been slightly modified:

#### 4.1. The normal force model

The *MercuryDPM* contact model `HertzianViscoelastic` applies a Hertzian normal force between two particles, given by

$$F^n = k^n \delta^n - \gamma^n v_{rel}^n, \quad (4.1)$$

where the normal stiffness  $k^n$  and normal dissipation coefficient  $\gamma^n$  are

$$k^n = \frac{4}{3} E^* \sqrt{R^* \delta_n}, \quad (4.2)$$

and

$$\gamma^n = \beta^n \sqrt{M^* k^n}. \quad (4.3)$$

---

<sup>1</sup>Note, this will shortly change to 1.0.Beta as the alpha test of this new release is almost complete.

Here,  $\delta^n$  is the normal overlap,  $v_{rel}^n$  the relative velocity in normal direction,  $E^*$  the effective Young's modulus,  $R^*$  the effective radius and  $M^*$  the effective mass.  $\beta^n$  is the normal damping factor which is related to the restitution coefficient  $\epsilon$  as follows,

$$\beta^n = \frac{\sqrt{5} \ln \epsilon}{\sqrt{\ln^2 \epsilon + \pi^2}}. \quad (4.4)$$

*Change:* Previously, the user directly specified the effective elastic modulus  $E^*$  using the command `setElasticModulus`. This caused confusion because most users expect to specify instead Young's modulus  $E$ , and Poisson's ratio,  $\nu$ , and compute the effective elastic modulus as  $E^* = \frac{1}{2}E/(1-\nu)$ . Thus, we replaced the function `setElasticModulus` with `setEffectiveElasticModulus` and added a helper function, `computeEffectiveElasticModulus`, to compute  $E^*$  from  $E$  and  $\nu$ .

#### 4.2. The tangential force model

The *MercuryDPM* contact model `Mindlin` applies a simplified version of the Mindlin model of sliding friction, as proposed by Di Renzo and Di Maio [15], where the tangential force is given by

$$F^t = \frac{2}{3}k^t\delta^t - \gamma^t v_{rel}^t, \quad (4.5)$$

where the tangential stiffness  $k^t$  and tangential dissipation coefficient  $\gamma^t$  are determined as

$$k^t = 8G^*\sqrt{R^*\delta^n}, \quad (4.6)$$

and

$$\gamma^t = \beta^t \sqrt{M^*k^t}. \quad (4.7)$$

In these equations,  $\delta^t$  is the tangential overlap,  $v_{rel}^t$  is the relative velocity in tangential direction,  $G^*$  is the effective shear modulus and  $\beta^t$  is the tangential damping coefficient.

*Change:* The dissipation coefficient in Eq. (4.7) previously read  $\gamma^t = \beta^t \sqrt{8M^*k^t}$ . The factor 8 in this expression was removed to allow for the following more compact notation of the tangential damping factor

$$\beta^t = \sqrt{\frac{2}{3}}\beta^n. \quad (4.8)$$

Note:  $\beta^t$  is currently an input parameter that the user must set. In the future, a  $\beta^t$  will be set to a default value according to Eq. (4.8), and can be overridden by the user. This will change for the 2.x series.

#### 4.3. The rolling friction model

The *MercuryDPM* contact model `MindlinRollingTorsion` applies a rolling friction according to Luding's model [14]: When a rolling friction coefficient is specified by the user, a corresponding rolling torque is applied, which is calculated as

$$T^{ro} = R^* F^{ro} \quad (4.9)$$

where  $F^{ro}$  is the rolling torsion force.

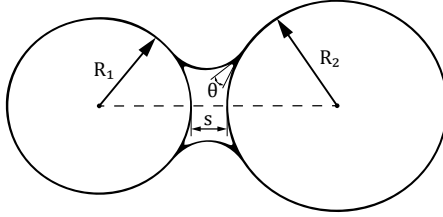


FIGURE 1. Sketch of a liquid bridge between two unequal-sized particles of radius  $R_1$  and  $R_2$ . Here,  $\theta$  is the contact angle, a material property, and  $s$  the separation distance between the two particles.

*Change:* In previous versions, the effective diameter was used as a prefactor in (4.9), rather than the effective radius, for calculating the rolling torque. This model has been updated to correctly satisfy Luding’s model [14].

## 5. Liquid migration model of Lian and Seville

The majority of research on granular media has concentrated on dry granular materials. However, in industry and nature, we often encounter wet granular materials. An example currently being investigated with *MercuryDPM* is the collision between wet (sand) particles, including moist sediment transport by wind and pneumatic conveying. Wet granular materials are cohesive due to surface tension, which is a significant distinction between dry and wet granular materials.

The *MercuryDPM* contact models `LiquidBridgeWillet` and `LiquidMigrationWillet` already have an implementation of liquid bridges. These are based on work of Willet et al. [16], which was developed by experimental data where forces due to liquid bridges between particles were measured.

We have now added a second model, `LiquidMigrationLS`, which is based on work of Lian and Seville [17], which is an extension of Willet’s model to include a lubrication viscous force. The viscous force is assumed to be the dominant force during the contact, induced by the squeezing out and pulling in of the liquid. The model was validated in particle simulations against collision experiments between a wet particle and a flat surface by Zhang and Wu [18], and is applicable to describe the dynamic interaction between wet particles; whereas, the previous model was only strictly correct for static (or slow moving) scenarios. Figure 1 shows the capillary liquid bridge between two particles of different sizes.

This code is available in the master version of *MercuryDPM* and is best demonstrated by `TwoParticleElasticCollisionInteractionWithLiquidMigrationLSSelfTest.cpp` in the source directory `Drivers/SelfTests/Interactions`.

It keeps the liquid migration features of Willet’s model, but differs in the force calculation: The capillary force is based on the half-filling angle which depends on the value of the contact angle and liquid bridge volume. The viscous force exists both in normal and tangential directions of potential contact, and is active only between a limited separation distance (smaller than rupture distance) and the rupture distance. Its magnitude depends on the relative velocity of the particles with an interstitial liquid bridge.

Within the `LiquidMigrationLS` model, one can define the liquid viscosity, based on the liquid type they model through `setViscosity()`. Similar to `LiquidMigrationWillet`, one also needs to define the volume of the liquid carried by the particles and the maximum and minimum value of the liquid bridge volume. In Figure 2, particles connected by liquid bridges are visualised in Paraview.

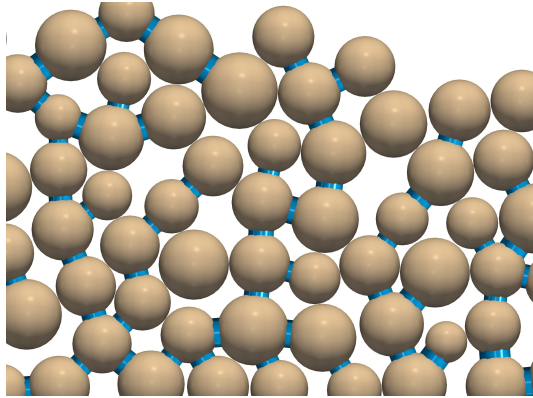


FIGURE 2. Wet (sand) particles connected by liquid bridges, using the `LiquidMigrationLS` contact model.

## 6. Spray modelling

As stated above, many industrial applications include wet granular materials; often, the moisture is added by a spray. In *MercuryDPM*, the `DropletBoundary` has been implemented, which can insert liquid droplets to wet the particles during a simulation. The implementation is simple: Each time step, droplets get inserted according to a function specified by the user. Droplets have a mass and velocity, and move under the influence of gravity. They do not interact with each other, but if a droplet contacts a particle, the particle absorbs the liquid and the droplet is removed. Note, as walls currently cannot absorb liquid, the liquid is lost when the droplet contacts a wall.

The user can define freely how droplets are inserted via the function `setGenerateDroplets`; one can e.g. set the rate of insertion, the droplet radii, the position of insertion (nozzle position and angle), and the velocity

of insertion (spray angle and droplet speed). For example, the user can add a droplet boundary that acts as a spray nozzle. In figure 3, we show one of the most common nozzles for spraying droplets, a flat fan spray nozzle.

This code is available in version 1.x of *MercuryDPM* and the figure is created by `NozzleDemo.cpp` in the source directory `Drivers/Demos/IndustrialMixers`.

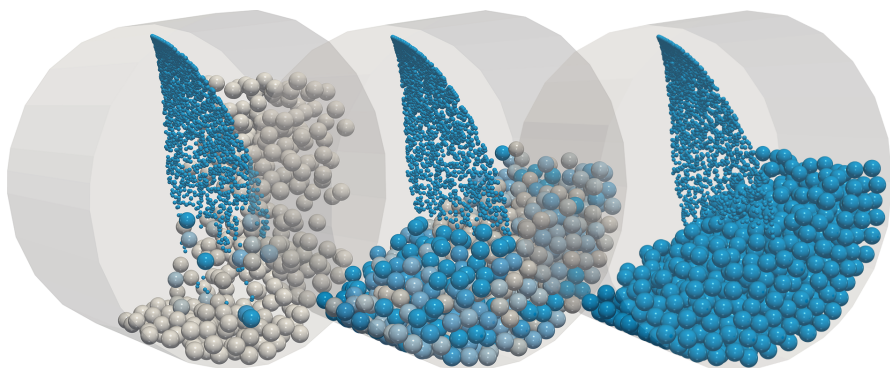


FIGURE 3. Wetting of particles by a flat-fan spray nozzle in a rotating drum. Time increasing from left to right. Small spheres are droplets, large spheres are particles. Particle colour (grey to blue) indicates increasing liquid content.

## 7. Dealing with true particle size

The `InsertionBoundary` in *MercuryDPM* was reworked for version 1.x. It is now capable of inserting particle mixtures composed of a variety of different materials and particle size distributions (PSD) into a single insertion volume. Moreover, the new interface accepts different types of PSD functions, based on volume, length, surface area or number of particles. This will allow for a more accurate representation of particle mixtures (see Figure 4), which is crucial for several applications and phenomena, such as segregation and mixing. Figure 4 is created by a script comparing a cumulative number PSD to the same one inserted by the driver code `PSDSelfTest.cpp`. The script is available in the `master`.

Furthermore, polydisperse particle packings used to be insertable only into cubic volumes. However, several applications require the insertion of particles into more complex geometries. Therefore, the insertion routine was reworked to allow the geometry of the fill volume and the properties of inserted particles to be set independently. It is now straightforward to create complex insertion regions. For a new insertion boundary object, only the geometry has to be defined.

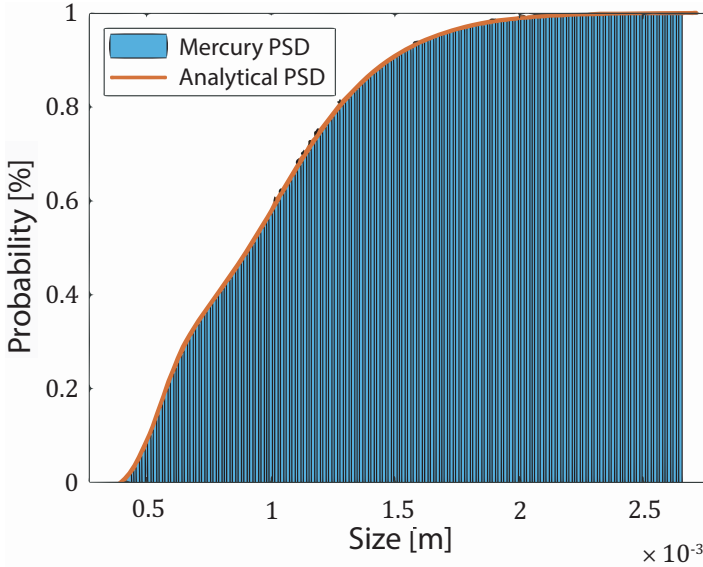


FIGURE 4. Cumulative PSD of inserted particles, compared to the input PSD.

## 8. Dealing with true particle shape - rigid clumps in MercuryDPM

Rigid clumps of spherical particles are an important tool to analyse the behaviour of granular materials consisting of particles of irregular shapes with the discrete element method. By *rigid clump* (just *clump*, or *multiparticle*) we imply the aggregate of  $N$  rigid (spherical) particles of a given density, that are rigidly linked to each other at given relative translational and rotational positions. Note, currently the code has only been demonstrated for spherical particles but in principle you should be able to form rigid clumps from more general super-quadric particles. The constituent particles of a clump will be referred to as *pebbles*. The pebbles may (or may not) have overlaps, introducing volumes within a clump that belong to more than one pebble. The contact detection algorithm treats contacts and corresponding forces/moments, as well as the forces arising from the force fields at the pebble level, while the dynamics computation algorithm treats a clump as a single rigid body, that is accelerated by the resultant force/moment, properly summed up over the pebbles.

The rigid clump function in *MercuryDPM* is implemented as a multi-level structure. A third-party library CLUMP [19] is used to generate positions and radii of pebbles that describe the given nonspherical shape (Figure (5) (A)). The CLUMP tool provides pebble data, which, along with the optionally provided initial STL-format shape of the clump, constitute an input of the MClump pre-processing tool (part of *MercuryDPM*). It centres and



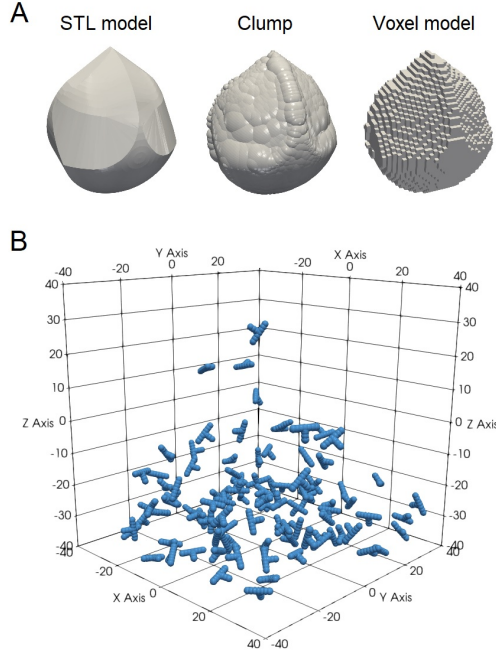


FIGURE 5. (A) STL model of a complex particle shape, the corresponding rigid clump, generated by [19] and the voxel model. (B) Sample simulation - 100 nonspherical particles in a gravity field, within a box with fully elastic walls.

rotates the clump, aligning its principal axes with global Cartesian axes, and computes clump's tensor of inertia using the prescribed algorithm (summation over (non-overlapping) pebbles, summation over voxels, summation over tetrahedrons using STL representation (Figure (5) (B))). A special header library for the driver file introduces necessary modifications of *MercuryDPM* virtual members, enabling clump dynamics. The driver file loads the list of clump instances generated by *MClump*, and, using them, generates necessary distributions of nonspherical particles and computes their dynamical evolution.

Figure (5) (B) highlights an example of using nonspherical particles in *MercuryDPM* (driver file `Drivers/MultiParticle/BulkTs.cpp`). The rigid clump feature is currently available in the master (developer's version), and will be included in future releases of *MercuryDPM* and is best demonstrated by the codes in the directory `Drivers/MultiParticle`.

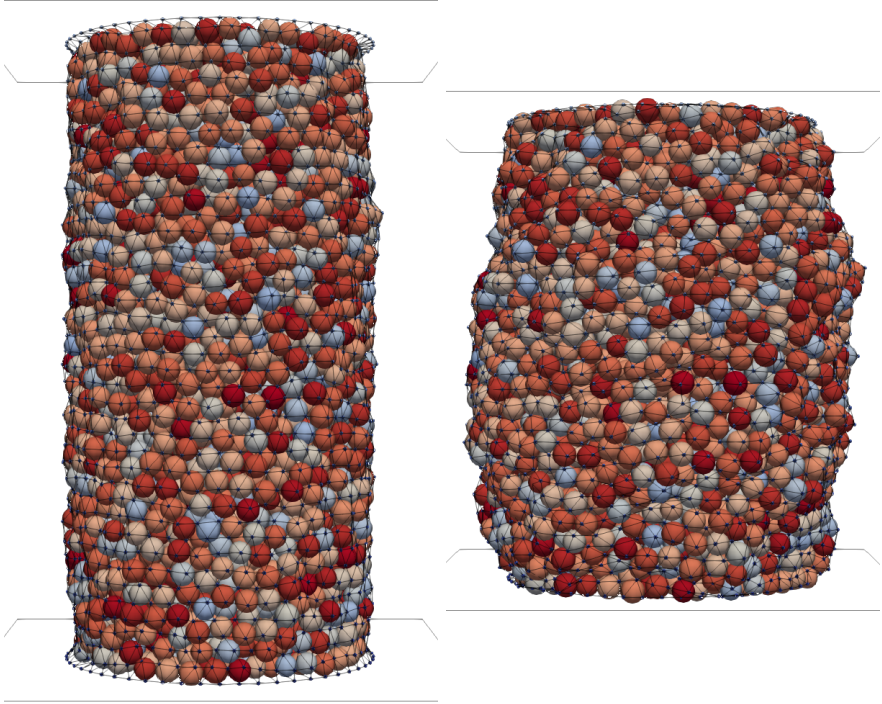


FIGURE 6. A triaxial test cell in its initial and final state simulated with the implemented mass spring system.

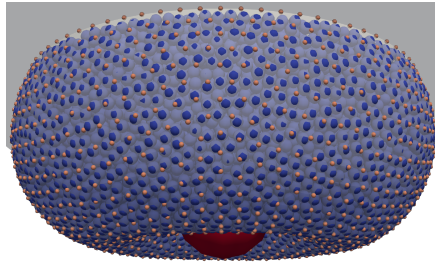


FIGURE 7. A snapshot from a granular gripper simulation executed with the implemented mass spring system.

## 9. Membranes

Interactions between membranes and granular particles may occur in systems like triaxial tests but also in applications from the soft robotics community, such as the granular gripper.

One possibility to represent membranes within granular simulations is a mass-spring system. Due to its particle-based nature it is easily integrated in

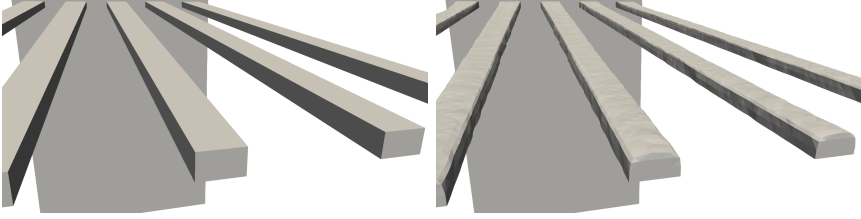


FIGURE 8. Before (left) and after (right) images for the wear on a vibration screen after a granular flow was passed over.

discrete particle simulations. Depending on the chosen mesh and the nature of the spring, it is possible to approximate all material properties [20].

*MercuryDPM* version 1.x now supports the use of mass-spring systems containing distance springs for the in-plane dynamics and a bending force penalty originating from cloth simulations [21]. If the connectivity of the masses is given by a triangular mesh with a hexagonal unit cell, the spring constant of the distance springs may be calculated with  $k = E \frac{3t}{2\sqrt{3}}$ , where  $t$  is the thickness and  $E$  the elastic modulus of the membrane [22, 23]. Note that the simplicity of this setup restricts us to a Poisson's ratio of  $\nu = 1/3$ .

In *MercuryDPM* the mass-spring system is constructed by inserting particles in place of the masses and connecting these according to a given mesh connectivity, which may be specified via a STL file or given directly in the code. As the connectivity has to be specified, the implementation does not rely on any contact detection algorithm. Although the code supports the usage of the vertex particles to detect contacts between the membrane and the granulate, as it was originally introduced by de Bono et al. [24] for triaxial test simulations, it is advised to deactivate these by a proper choice of species. In that case, the contacts are instead calculated using additional triangular wall elements inserted between the vertex particles. The position and dynamics of these wall elements are purely defined by the vertex particles. Quantities such as local velocities and the calculated contact forces are transferred between the triangles and the vertex particles using barycentric interpolation.

Figure 6 shows an example of the triaxial shear test and is created by a private driver code. Figure 7 shows the granular gripper application from the published paper [25]. The membrane is best demonstrated by the code `MembraneDemo.cpp` in the directory `Drivers/Demos/Membrane`.

## 10. Modelling surface wear

Often in industry, granular flows lead to the wearing of surfaces over time due to abrasion, causing apparatus to misperform or even fail. One such apparatus is a vibrating screen, used to grade materials in various industrial applications including steel making. The left panel in Figure 8 shows a typical vibrating screen from the front. Grains are dropped at the back of the screen

and slide forward, falling through the increasingly widening gaps. As the material slides across, the surface is worn down, increasing the size of the particles that are able to fit through the screen. This feature will be available in future releases.

In *MercuryDPM*, the Reye–Archard–Khrushchov wear model has been implemented [26], which states that the volume of material removed is proportional to the work done by the friction forces. The proportional is normally set to higher values than for the real material in order to accelerate the process and reduce the simulation time.

Currently, only for triangulated walls, the wear model can be used to update the surface and hence, predict the long term effect of particle abrasion on walls. The process is summarised as follows:

1. Compute the (local) frictional forces on each wall segment.
2. For each triangle compute the (local) volume to be removed via the Reye–Archard–Khrushchov model.
3. Iteratively move the mesh vertices to match each local volume of removed material.

In Figure 8 you can see the shape of a vibrating screen before (left) and after (right) a granular material has flowed over it.

## 11. Soft Kill

*MercuryDPM* now has an internal signal handler, which allows us to control what happens when external signals are received by the code. This enables the code to be run on resources with limited time available, as it will allow the code to be stopped (and restarted) automatically based on external triggers. Thus, code can be deployed on cheap, unused, temporary computing resources like AWS spot instances.

The signal handler treats various interrupt signals that are sent to the processors. A common example is SIGINT, which can be sent by the user to a running program by pressing CTRL+C. This interrupt is caught by the signal handler, triggering a soft-kill routine: instead of stopping the code immediately, this feature allows *MercuryDPM* simulations to continue until all operations necessary to write the restart file are finished. This way the output data (e.g. \*.restart, \*.fstat, \*.data) will be consistent (same timestep) and complete (no partly written files). The simulation can thus be restarted safely at a later time.

Currently this feature catches the following signals, and triggers the actions described:

- SIGINT (signal 2): This signal is sent by CTRL+ C. Soft kill continues the simulation until the current time step is completed. Then it forces *MercuryDPM* to write to all output files, in particular the restart file.
- SIGTERM (signal 15): This is generated in linux by kill. It triggers the same action as SIGINT.

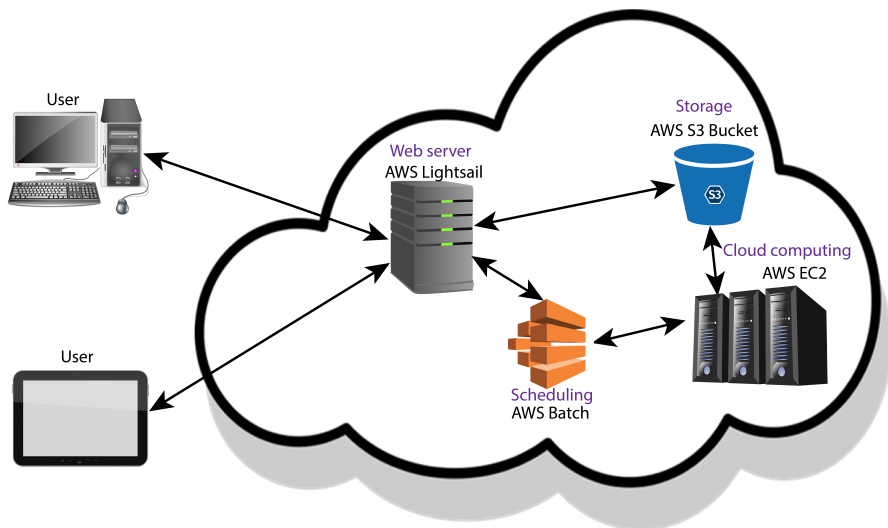


FIGURE 9. Diagram showing the current cloud architecture.

- **SIGKILL (signal 9):** This is generated in linux by `kill -9`. It terminates *MercuryDPM* immediately. In other words, this signal stops the soft-kill feature.

Note: Using tools like `htop` and `F9` in Linux, the user can send specific signals or kill commands to *MercuryDPM*. Also, one can use `$kill -SIGNAL PID`, where `SIGNAL` is the signal number listed above and `PID` is the process id of the *MercuryDPM* executable.

## 12. Mercury Cloud

*MercuryDPM* also has its own official spin-off company, *MercuryLab*, which aims to facilitate access to the advanced features of *MercuryDPM* for industry and academia. *MercuryLab* often develops new features for industrial clients, but always returns them to the open-source repository. The company is currently one of the biggest contributors to the code base.

One of the developments of *MercuryLab* is its AWS-deployed cloud interface for the code, *MercuryCloud* (<https://cloud.mercurylab.org/>). This interface allows scalable, GUI-based, on-demand access to (certain) features of the code via an internally developed web interface. The architecture of the cloud is shown in figure 9 and is available from any device with internet access and a web browser.

*MercuryCloud* is still in development, thus its capabilities are limited, but growing. Academic users are able to access the facilities at a considerable discount. Utilising the soft-stop feature described in the previous section, we hope to make use of cheap AWS spot instances, which are often at least 60% cheaper.

### 13. Funding acknowledgements

*MercuryDPM* has been supported by many projects, both past and present. The features presented here were (partially) funded by the following grants:

The Dutch Research Council (NWO), in the framework of the ENW PPP Fund for the topsectors and from the Ministry of Economic Affairs in the framework of the “PPS-Toeslagregeling”; NWO-OTP grant 15050 “Multiscale modelling of agglomeration”; and NWO-VIDI grant 16604 “Virtual prototyping of particulate processes”.

We also gratefully acknowledge funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project number 411517575.

Finally, great thanks also goes to the Dutch Sectorplan Engineering.

### References

- [1] T. Weinhart, D.R. Tunuguntla, M.P. Van Schrojenstein-Lantman, A.J. Van Der Horn, I.F.C. Denissen, C.R. Windows-Yule, A.C. De Jong, and A.R. Thornton. MercuryDPM: A fast and flexible particle solver part A: Technical advances. In *Springer Proceedings in Physics*, volume 188, 2017.
- [2] T. Weinhart, D.R. Tunuguntla, M.P.V.S. Lantman, I.F. Denissen, R.W.-Y. Christopher, H. Polman, J.M.F. Tsang, B. Jin, L. Orefice, K.V.D. Vaart, S. Roy, H. Shi, A. Pagano, W. DenBreeijen, B.J. Scheper, A. Jarray, S. Luding, and A.R. Thornton. MercuryDPM: Fast, flexible particle simulations in complex geometries part II: Applications. In *5th International Conference on Particle-Based Methods - Fundamentals and Applications, PARTICLES 2017*, 2017.
- [3] A. R. Thornton, M. Post, L. Orefice, P. Rapino, S. Roy, H. Polman, M. Y. Shaheen, J. E. A. Naranjo, H. Cheng, L. Jing, et al. Faster, more flexible, particle simulations: The future of MercuryDPM. In *8th International Conference on Discrete Element Methods, DEM 2019*, 2019.
- [4] A. R. Thornton, D. Krijgsman, A. te Voortwis, V. Ogarko, S. Luding, R. Fransen, S. Gonzalez, O. Bokhove, O. Imole, and T. Weinhart. A review of recent work on the discrete particle method at the University of Twente : An introduction to the open-source package MercuryDPM. In *DEM 6: 6<sup>th</sup> International Conference on Discrete Element Methods and Related Techniques*, pages 393–399, 2013.
- [5] A. R. Thornton, D. Krijgsman, R. H. A. Fransen, S. Gonzalez, D. R. Tunuguntla, A. te Voortwis, S. Luding, O. Bokhove, and T. Weinhart. MercuryDPM: fast particle simulations in complex geometries. *EnginSoft newsletter simulation based engineering & sciences*, 10(1):48–53, 2013.
- [6] D. R. Tunuguntla, T. Weinhart, and A. R. Thornton. Discrete particle simulations with MercuryDPM. In *Alert Doctoral School 2017 : Discrete Element Modeling*, 2017.
- [7] T. Weinhart, L. Orefice, M. Post, M. P. van Schrojenstein Lantman, I. F. C. Denissen, D. R. Tunuguntla, J. M. F. Tsang, H. Cheng, M. Y. Shaheen, H. Shi, et al. Fast, flexible particle simulations – an introduction to MercuryDPM. *Computer Physics Communications*, 249:107129, 2020.

- [8] V. Ogarko and S. Luding. A fast multilevel algorithm for contact detection of arbitrarily polydisperse objects. *Comp. Phys. Comm.*, 183:932–936, 2012.
- [9] D. R. Tunuguntla, A. R. Thornton, and T. Weinhart. From discrete elements to continuum fields: Extension to bidisperse systems. *Computational Particle Mechanics*, 3(3):349–365, 2016.
- [10] M. Heil and A. L. Hazel. oomph-lib—an object-oriented multi-physics finite-element library. In *Fluid-structure interaction*, pages 19–49. Springer, 2006.
- [11] H. Cheng, A. R. Thornton, S. Luding, A. L. L Hazel, and T. Weinhart. Concurrent multi-scale modeling of granular materials: Role of coarse-graining in fem-dem coupling. *Computer Methods in Applied Mechanics and Engineering*, 2022.
- [12] J. Ahrens, B. Geveci, and C. Law. Paraview: An end-user tool for large data visualization, 2005.
- [13] R. D. Mindlin and H. Deresiewicz. Elastic spheres in contact under varying oblique forces. *Journal of Applied Mechanics*, 20:327–344, 9 1953.
- [14] S. Luding. Cohesive, frictional powders: contact models for tension. *Granular Matter*, 10(4):235–246, 2008.
- [15] A. Di Renzo and F. P. Di Maio. Comparison of contact-force models for the simulation of collisions in DEM-based granular flow codes. *Chemical Engineering Science*, 59:525–541, 2004.
- [16] C. D. Willett, M. J. Adams, S. A. Johnson, and J. P. K. Seville. Capillary bridges between two spherical bodies. *Langmuir*, 16(24):9396–9405, 2000.
- [17] G. Lian and J. Seville. The capillary bridge between two spheres: New closed-form equations in a two century old problem. *Advances in colloid and interface science*, 227:53–62, 2016.
- [18] L. Zhang and C. Wu. Discrete element analysis of normal elastic impact of wet particles. *Powder Technology*, 362:628–634, 2020.
- [19] V. Angelidakis, S. Nadimi, M. Otsubo, and S. Utili. Clump: A code library to generate universal multi-sphere particles. *SoftwareX*, 15:100735, 2021.
- [20] M. Kot and H. Nagahashi. Mass spring models with adjustable Poisson’s ratio. *Vis Comput*, 33(3):283–291, 2017.
- [21] R. Bridson, S. Marino, and R. Fedkiw. Simulation of clothing with folds and wrinkles. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA ’03, pages 28–36. Eurographics Association, 2003.
- [22] M. Ostoja-Starzewski. Lattice models in micromechanics. *Applied Mechanics Reviews*, 55(1):35–60, 2002.
- [23] T. Qu, Y. T. Feng, Y. Wang, and M. Wang. Discrete element modelling of flexible membrane boundaries for triaxial tests. *Computers and Geotechnics*, 115:103154, 2019.
- [24] J. de Bono, G. Mcdowell, and D. Wanatowski. Discrete element modelling of a flexible membrane for triaxial testing of granular material at high pressures. *Géotechnique Letters*, 2(4):199–203, 2012.
- [25] H. Götz, A. Santarossa, A. Sack, T. Pöschel, and P. Müller. Soft particles reinforce robotic grippers: Robotic grippers based on granular jamming of soft particles. *Granular Matter*, 24(1):31, 2022.

- [26] J. Archard. Contact and rubbing of flat surfaces. *Journal of Applied Physics*, 24(8):981–988, 1953.

Anthony R. Thornton  
University of Twente  
Department of Thermal and Fluid Engineering  
The Netherlands  
and  
MercuryLab B.V.  
The Netherlands  
e-mail: [a.r.thornton@utwente.nl](mailto:a.r.thornton@utwente.nl)

Timo Plath  
University of Twente  
Department of Thermal and Fluid Engineering  
The Netherlands  
e-mail: [t.plath@utwente.nl](mailto:t.plath@utwente.nl)

Igor Ostanin  
University of Twente  
Department of Thermal and Fluid Engineering  
The Netherlands  
e-mail: [i.ostanin@utwente.nl](mailto:i.ostanin@utwente.nl)

Holger Götz  
Institute for Multiscale Simulations  
Friedrich-Alexander-Universität Erlangen-Nürnberg  
Cauerstraße 3, 91058 Erlangen  
Germany  
e-mail: [holger.goetz@fau.de](mailto:holger.goetz@fau.de)

Jan-Willem Bisschop  
University of Twente  
Department of Thermal and Fluid Engineering  
The Netherlands  
and  
MercuryLab B.V.  
The Netherlands  
e-mail: [j.bisschop@mercurylab.org](mailto:j.bisschop@mercurylab.org)

Mohamed Hassan  
University of Twente  
Department of Thermal and Fluid Engineering  
The Netherlands  
e-mail: [m.b.a.hassan@utwente.nl](mailto:m.b.a.hassan@utwente.nl)

Raïsa Roeplal  
Delft University of Technology  
Department of Maritime and Transportation Technology  
The Netherlands  
e-mail: [r.n.roeplal@tudelft.nl](mailto:r.n.roeplal@tudelft.nl)



Xiuqi Wang  
University of Twente  
Department of Civil Engineering and Management  
The Netherlands  
e-mail: [x.wang-8@utwente.nl](mailto:x.wang-8@utwente.nl)

Sahar Pourandi  
University of Twente  
Department of Thermal and Fluid Engineering  
The Netherlands  
e-mail: [s.pourandi@utwente.nl](mailto:s.pourandi@utwente.nl)

Thomas Weinhart  
University of Twente  
Department of Thermal and Fluid Engineering  
The Netherlands  
and  
MercuryLab B.V.  
The Netherlands  
e-mail: [t.weinhart@utwente.nl](mailto:t.weinhart@utwente.nl)