Computer Physics Communications 249 (2020) 107129

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

CPC 50th anniversary article

Fast, flexible particle simulations — An introduction to MercuryDPM☆,☆☆

Thomas Weinhart [a,b,*], Luca Orefice [c,d], Mitchel Post [a], Marnix P. van Schrojenstein Lantman [a], Irana F.C. Denissen [a], Deepak R. Tunuguntla [a], J.M.F. Tsang [e], Hongyang Cheng [a], Mohamad Yousef Shaheen [a], Hao Shi [a,b], Paolo Rapino [b], Elena Grannonio [g], Nunzio Losacco [g], Joao Barbosa [h], Lu Jing [f], Juan E. Alvarez Naranjo [a], Sudeshna Roy [a], Wouter K. den Otter [a], Anthony R. Thornton [a,b]

[a] Multiscale Mechanics, Engineering Technology, MESA+, University of Twente, PO Box 217, 7500 AE Enschede, The Netherlands
[b] Mercury Lab BV, Mekkelholtsweg 10, 7523 DE Enschede, The Netherlands
[c] Research Center Pharmaceutical Engineering (RCPE) GmbH, Inffeldgaße 13, 8010 Graz, Austria
[d] European Consortium on Continuous Pharmaceutical Manufacturing (ECCPM), 8010 Graz, Austria
[e] DAMTP, Centre for Mathematical Sciences, University of Cambridge, Wilberforce Road, Cambridge CB3 0WA, United Kingdom
[f] Department of Chemical and Biological Engineering, Northwestern University, Evanston, IL 60208, USA
[g] Department of Civil Engineering and Computer Science, University of Rome "Tor Vergata", Via del Politecnico 1, 00133 Rome, Italy
[h] Department of Engineering Structures, Section of Dynamics of Solids and Structures, CiTG, TU Delft, Stevinweg 1, 2628 CN Delft, The Netherlands

ARTICLE INFO

Article history:
Received 12 August 2019
Received in revised form 29 November 2019
Accepted 2 December 2019
Available online 27 December 2019

Keywords:
Granular materials
DEM
DPM
MercuryDPM
Open-source

ABSTRACT

We introduce the open-source package *MercuryDPM*, which we have been developing over the last few years. *MercuryDPM* is a code for discrete particle simulations. It simulates the motion of particles by applying forces and torques that stem either from external body forces, (gravity, magnetic fields, etc.) or particle interactions. The code has been developed extensively for granular applications, and in this case these are typically (elastic, plastic, viscous, frictional) contact forces or (adhesive) short-range forces. However, it could be adapted to include long-range (molecular, self-gravity) interactions as well.

MercuryDPM is an object-oriented algorithm with an easy-to-use user interface and a flexible core, allowing developers to quickly add new features. It is parallelised using MPI and released under the BSD 3-clause licence. Its open-source developers' community has developed many features, including moving and curved walls; state-of-the-art granular contact models; specialised classes for common geometries; non-spherical particles; general interfaces; restarting; visualisation; a large self-test suite; extensive documentation; and numerous tutorials and demos. In addition, *MercuryDPM* has three major components that were originally invented and developed by its team: an advanced contact detection method, which allows for the first time large simulations with wide size distributions; curved (non-triangulated) walls; and multicomponent, spatial and temporal coarse-graining, a novel way to extract continuum fields from discrete particle systems. We illustrate these tools and a selection of other *MercuryDPM* features via various applications, including size-driven segregation down inclined planes, rotating drums, and dosing silos.

Program summary

Program Title: *MercuryDPM*
Program Files doi: http://dx.doi.org/10.17632/n7jmdrdc52.1
Licensing provisions: BSD 3-Clause
Programming language: C++, Fortran
Supplementary material: http://mercurydpm.org
Nature of problem: Simulation of granular materials, i.e. conglomerations of discrete, macroscopic particles. The interaction between individual grains is characterised by a loss of energy, making the behaviour of granular materials distinct from atomistic materials, i.e. solids, liquids and gases.

☆ The review of this paper was arranged by Prof. N.S. Scott.

☆☆ This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

* Corresponding author at: Multiscale Mechanics, Engineering Technology, MESA+, University of Twente, PO Box 217, 7500 AE Enschede, The Netherlands.
E-mail address: t.weinhart@utwente.nl (T. Weinhart).

https://doi.org/10.1016/j.cpc.2019.107129

Solution method: *MercuryDPM* (Thornton et al., 2013, 2019; Weinhart et al., 2016, 2017, 2019) is an implementation of the Discrete Particle Method (DPM), also known as the Discrete Element Method (DEM) (Cundall and Strack, 1979). It simulates the motion of individual particles by applying forces and torques that stem either from external forces (gravity, magnetic fields, etc.) or from particle-pair and particle–wall interactions (typically elastic, plastic, dissipative, frictional, and adhesive contact forces). DPM simulations have been successfully used to understand the many unique granular phenomena – sudden phase transitions, jamming, force localisation, etc. – that cannot be explained without considering the granular microstructure.

Unusual features: *MercuryDPM* was designed *ab initio* with the aim of allowing the simulation of realistic geometries and materials found in industrial and geotechnical applications. It thus contains several bespoke features invented by the *MercuryDPM* team: *(i)* a neighbourhood detection algorithm (Krijgsman et al., 2014) that can efficiently simulate highly polydisperse packings, which are common in industry; *(ii)* curved walls (Weinhart et al., 2016) making it possible to model real industrial geometries exactly, without triangulation errors; and *(iii)* MercuryCG (Weinhart et al., 2012, 2013, 2016; Tunuguntla et al., 2016), a state-of-the-art analysis tool that extracts local continuum fields, providing accurate analytical/rheological information often not available from experiments or pilot plants. It further contains a large range of contact models to simulate complex interactions such as elasto-plastic deformation (Luding, 2008), sintering (Fuchs et al., 2017), melting (Weinhart et al., 2019), breaking, wet and dry cohesion (Roy et al., 2016, 2017), and liquid migration (Roy et al., 2018), all of which have important industrial applications.

1. Introduction

Granular materials – conglomerations of discrete, macroscopic particles – are ubiquitous in both industry and nature. They range from natural materials like snow, sand, soil, coffee, rice and coal to man-made agglomerates such as medicinal tablets, catalysts or animal feed. Understanding the behaviour of granular media is of paramount importance to the pharmaceutical, mining, food processing and manufacturing industries, and highly relevant to the prediction and prevention of landslides, earthquakes and other geophysical phenomena.

MercuryDPM [1–4] is an open-source package for simulating granular materials with the discrete particle method (DPM) [5]. It simulates the motion of N particles in a system constrained by N_w walls and body forces. It assumes:

(i) Particles are unbreakable; however, breakage can be included by forming clusters of 'primary' particles, see Section 6.3 for details.
(ii) Particles are undeformable, such that the particle masses m_i and inertia tensor $\mathbf{I}_i$ are constant in the body-based frame. Note that clusters can be used to model deformable particles, see Section 6.3.
(iii) All interactions between the particles are binary, i.e. all internal forces/torques are due to particle pair interactions.
(iv) Each particle pair i, j has at most a single contact point $\mathbf{c}_{ij}$ at which the interaction forces $\mathbf{f}_{ij}$ and torques $\boldsymbol{\tau}_{ij}$ act.
(v) All external forces/torques acting on a particle i are either body forces $\mathbf{f}_i^\mathrm{b}$ or interaction forces $\mathbf{f}_{ik}^\mathrm{w}$ with a wall k. The same is true for torques.

The force and torque acting on each particle i can then be computed as

$$\mathbf{f}_i = \sum_{j=1}^{N} \mathbf{f}_{ij} + \sum_{k=1}^{N_\mathrm{w}} \mathbf{f}_{ik}^\mathrm{w} + \mathbf{f}_i^\mathrm{b},$$

$$\boldsymbol{\tau}_i = \sum_{j=1}^{N} \mathbf{r}_{ij} \times \mathbf{f}_{ij} + \boldsymbol{\tau}_{ij} + \sum_{k=1}^{N_\mathrm{w}} \mathbf{r}_{ik} \times \mathbf{f}_{ik}^\mathrm{w} + \boldsymbol{\tau}_{ik}^\mathrm{w} + \boldsymbol{\tau}_i^\mathrm{b},$$

with the branch vector $\mathbf{r}_{ij} = \mathbf{c}_{ij} - \mathbf{r}_i$ connecting the particle position $\mathbf{r}_i$ with the contact point $\mathbf{c}_{ij}$. For given initial conditions, Newton's second law can then be used to evolve the particles' velocities $\mathbf{v}_i$, positions $\mathbf{r}_i$, angular velocities $\boldsymbol{\omega}_i$ and orientations $\mathbf{q}_i$:

$$\frac{\mathrm{d}\mathbf{v}_i}{\mathrm{d}t} = \frac{1}{m_i}\mathbf{f}_i, \quad \frac{\mathrm{d}\mathbf{r}_i}{\mathrm{d}t} = \mathbf{v}_i, \quad \frac{\mathrm{d}\boldsymbol{\omega}_i}{\mathrm{d}t} = \mathbf{I}_i^{-1}\boldsymbol{\tau}_i, \quad \frac{\mathrm{d}\mathbf{q}_i}{\mathrm{d}t} = \mathbf{C}(\mathbf{q}_i)\boldsymbol{\omega}_i.$$

For computational stability, the orientation is stored as a quaternion $\mathbf{q}_i \in \mathbb{R}^4$, which requires the use of a transformation matrix $\mathbf{C}(\mathbf{q}_i)$; see [6,7] for details.

The above differential equations are solved numerically using the Velocity-Verlet algorithm, which is symplectic (thus, energy is conserved in case of elastic forces) and second-order accurate. Using a higher-order accurate time integration scheme would not increase accuracy, because most DPM contact models are non-differentiable.

MercuryDPM is written mainly in object-oriented C++, using many modern features from C++11. We try to follow the latest developments in the C++ language; however, we also guarantee the release version will work on two-year old compilers. The latest version uses parts of the Fortran LAPACK library [8]. However, the parts we use have been incorporated into the MercuryDPM source code, thereby avoiding an external dependency; only a Fortran compiler is required. The code has an easy-to-use user interface and a flexible core, allowing developers to quickly add new features. It is parallelised using MPI and released under the BSD 3-clause licence. Thus, it can be used as part of closed-source derivatives, as long as the derived software acknowledges the *MercuryDPM* team.

We have tried (and hopefully succeeded) in making *MercuryDPM* easy to learn for new users. Its installation process is simple and the package includes many tutorials as well as example codes that demonstrate the package's features. It is also supplemented by a detailed reference manual (`docs.mercurydpm.org`). Users otherwise unfamiliar with C++ have found the project's coding style intuitive, allowing them to focus on modelling problems.

The code is being developed by a global network of researchers and in the last few years has received contributions from universities such as Cambridge, Stanford, EPFL, Birmingham, Strathclyde, Sydney, Northwestern, Rome, Delft and Manchester, as well as industry, such as MercuryLab in Enschede and RCPE and ECCPM in Graz. We encourage all *MercuryDPM* users to merge the features they develop into *MercuryDPM*, thus becoming *MercuryDPM* developers.

As the code is fully open-source, all features we develop can be accessed and reused freely for both non-commercial and commercial use. The open-source philosophy allows the code base

to grow quickly, and the open-source development reduces the amount of coding errors, as you get near-imminent feedback from other developers. Its open-source community has developed many features, including moving and curved walls; state-of-the-art granular contact models; specialised classes for common geometries; non-spherical particles; general interfaces; restarting; visualisation; a large self-test suite; extensive documentation; and numerous tutorials and demos. In the following, we review some of these features.

2. Coding philosophy

MercuryDPM is written in an object-oriented programming style, i.e. it uses classes to define objects: spherical particles are objects of type `SphericalParticle`; planar walls are of type `InfiniteWall`; and periodic boundaries are of type `PeriodicBoundary`. A myriad of other classes have been implemented and ready for use, many of which are described in following sections The user can also derive their own classes by inheriting from existing ones, adding extra functionality.

The clear and structured nature of *MercuryDPM* means it is quick and easy to develop new features; however, the level of C++ required is still demanding to some users. Therefore, we are developing a graphical interface, opening it up to a whole new set of users, both academic and industrial.

3. Major components

MercuryDPM has three major components that were originally developed by its team: *(i)* a contact detection algorithm [10] that can efficiently simulate highly polydisperse packings, which are common in industry; *(ii)* curved walls [3], making it possible to model real industrial geometries exactly, without triangulation errors; and *(iii)* MercuryCG [11–14], a state-of-the-art analysis tool that extracts local continuum fields, providing accurate analytical/rheological information often not available from experiments or pilot plants.

3.1. Contact detection

Contact detection – determining which particle pairs are in contact – is one of the most complex parts of any DPM algorithm and can consume the majority of the computational time, if it is not carefully implemented.

The most basic contact detection simply loops through all particle pairs; this algorithm is of quadratic complexity, $\mathcal{O}(N^2)$, where N is the number of particles in the simulation. Because the rest of the DPM algorithm is of linear complexity, $\mathcal{O}(N)$, such a contact detection would make large simulations prohibitively expensive. A more efficient contact detection algorithm is needed.

Most DPM algorithms use the *linked-cell algorithm* for contact detection [15], illustrated in Fig. 1 left: Particles are placed into a grid whose cell size is the diameter of the largest particle. Thus, particles can only be in contact with particles in the same or in a neighbouring cell, reducing the number of necessary checks. For (nearly) monodispersed simulations, this algorithm is of linear complexity, $\mathcal{O}(N)$, and thus sufficiently efficient. However, the order complexity increases to quadratic, $\mathcal{O}(N^2)$, for highly polydisperse simulations, because the cell size is based on the largest particle diameter.

MercuryDPM uses the *hierarchical grid (hGrid)* [9,10,16], an advanced contact detection method that uses several grids for different particle sizes, as shown in Fig. 1 middle. This contact method gives *MercuryDPM* its name: *hGridDPM* $\rightarrow$ *HgDPM* $\rightarrow$ *MercuryDPM*. By carefully selecting the number of levels and cell sizes, linear complexity of the algorithm can be guaranteed even for the most challenging particle size distributions. The effectiveness of the approach has been proven theoretically and in simulations [9]: for highly polydisperse situations, the hierarchical grid is two orders of magnitude quicker than the linked-cell algorithm, see Fig. 1 right. Furthermore, the CPU time varied only minimally when comparing monodisperse, bidisperse and polydisperse systems with the same number of particles and volume fraction [9]. *This feature allows for the first time large simulations with wide size-distributions.*

The hierarchical grid algorithm is made up of two phases: mapping and contact detection. In the first phase, all particles are mapped onto a grid level. In the second phase, the potential contact partners for every particle in the system are determined. Both phases are of linear complexity, and allow straightforward parallelisation.

The two- or three-dimensional hierarchical grid is a set of L regular grids with different cell sizes $s_1 < s_2 < \cdots < s_L$. Each particle p is mapped into a specific cell in the lowest level grid big enough to contain the particle. A new mapping is done before every time step, as this is cheaper than tracking changes and updating the map. It must be noted that the cell size s_h of each level h can be set independently, in contrast to contact detection methods which use a tree structure for partitioning the domain [17], where the cell sizes are taken as double the size of the previous lower level of hierarchy, hence $s_{h+1} = 2s_h$. The flexibility of independently choosing s_h allows one to select the optimal cell sizes according to the particle size distribution, further improving the performance of the simulations [9].

The contact detection is split into two steps, and the search is done by looping over all particles p and performing the first and second steps consecutively for each p. The first step is a contact search at the level of insertion, using the classical linked-cell method: the search is done in the cell onto which p is mapped, and in its neighbour (surrounding) cells. Only half of the surrounding cells are searched, to avoid testing the same particle pair twice. The second step is the *cross-level search*. For a given particle p, one searches for potential contacts only at levels of smaller grid size. This implies that the particle p will be checked only against the smaller ones, thus avoiding double checks for the same pair of particles. See [9] for details of the algorithm.

3.1.1. Application: Segregation in rotating drums

Segregation of grains by size is a scientifically interesting and industrially relevant problem. In industrial situations, size-distributions often range over orders of magnitude and are highly polydispersed; whereas academic studies often consider bidispersity with a factor of only 2–10 in size. One key reason for this discrepancy is computational cost. However, the hierarchical grid, at the heart *MercuryDPM*, is over three orders of magnitude faster for bidispersed mixtures with a size ratio of 100, and even faster for truly polydisperse packings; see [16] for details. Fig. 2 shows a simulation where each particle is a unique size and the ratio of small to largest radii is 100. This is visualised using ParaView and was run on a single core on a normal desktop computer in a few hours.

3.2. Curved walls

A distinguishing feature of *MercuryDPM* is its support of curved geometric surfaces, or *walls*. Many types of walls are implemented and ready for use, such as

- Flat walls (implemented in `InfiniteWall`),
- Convex polygons (2D) or polyhedra (3D) (`IntersectionOfWalls`),
- Conical and cylindrical shapes, created by rotating a polygon around an axis (`AxisymmetricIntersectionOfWalls`),

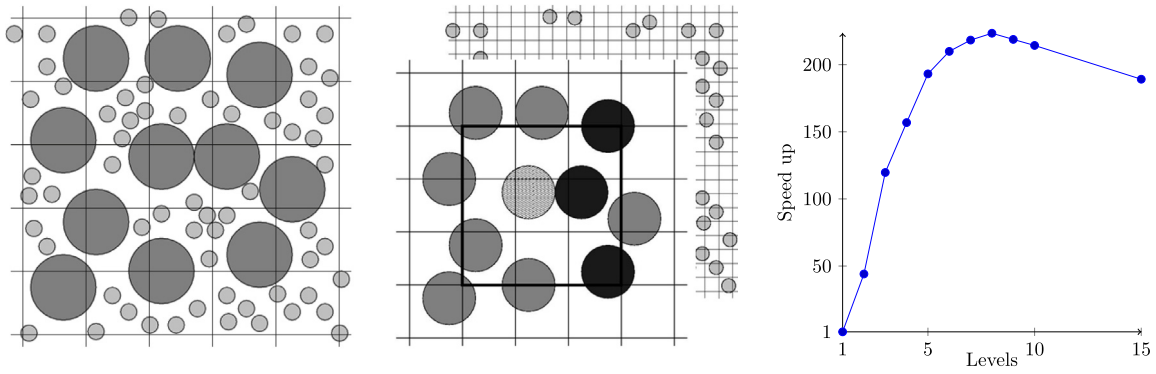

Fig. 1. Left: Linked-cell grid for contact detection for a bi-disperse system. Middle: A two-level hierarchical grid for the same case. Right: Speed-up factor for different numbers of levels for systems with a uniform volume distribution, $N = 125\,001$, $a_{max}/a_{min} = 50$.
Source: Data from [9].

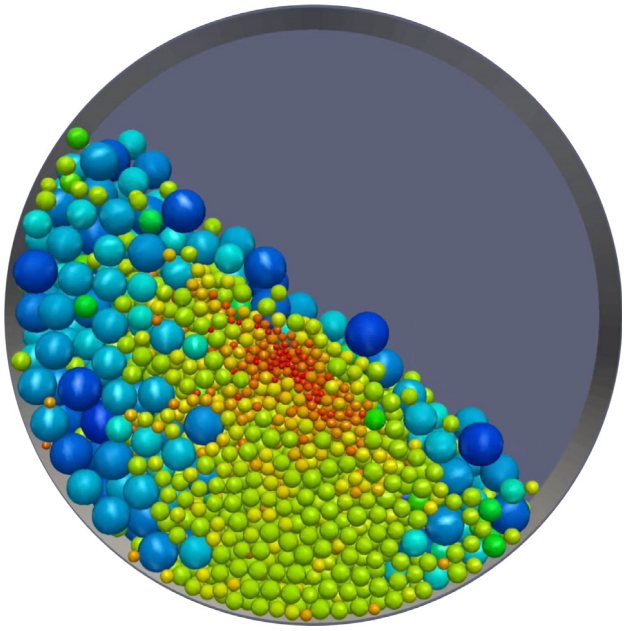

Fig. 2. Simulations visualised in ParaView of size-based segregation in drum. Colour indicates particle size, from red (small) to green (medium) to blue (large).

- Single- or double-threaded helixes (`Screw`),
- Coils [18] (`Coil`),
- Iso-surface of a piecewise linear function defined on a Cartesian grid [19] (`LevelSetWall`),
- NURBS surfaces [20] (`NurbsWall`).

See Fig. 3 for examples of the most common wall types.

Each wall k has a position $\mathbf{p}_k$ and orientation $\mathbf{q}_k$ that can be either a fixed value or a prescribed function, to simulate moving and rotating walls.

For contact detection, each wall type has a `getDistanceAndNormal(particle, distance, normal)` function that computes the contact normal $\mathbf{n}_{ij}$ and distance d from the wall for any given particle. These values are necessary to compute the contact force.

For many walls, the contact normal and distance is computed analytically (and thus efficiently): For example, for a flat wall k and a spherical particle i, the normal direction $\mathbf{n}_{ik}$ can be computed from the wall orientation, and the distance to the particle is given by $d = (\mathbf{p}_i - \mathbf{p}_k) \cdot \mathbf{n}_{ik}$. Analytic solutions are also used for `IntersectionOfWalls`, `AxisymmetricIntersectionOfWalls`, `TriangleWall`, `NURBSWall`, and `LevelSetWall`. Note that `IntersectionOfWalls` are not simple flat surfaces, but have defined face, edge and vertex contacts; Fig. 4 shows the normal and distance computed for the case of a triangular wall.

More complex wall shapes like the `Coil` or `Screw` require an iterative scheme. In both cases, Newton iteration is used to minimise the distance to the wall. Fig. 5 shows the normal and distance for a `Screw`.

In particular NURBS surfaces are very general and can be used to simulate many types of walls. However, if a particular surface type is not yet implemented, the user can define it by creating a new wall type and writing an appropriate `getDistanceAndNormal` function.

Most other codes approximate curved surfaces via triangulated walls. This can be done in *MercuryDPM* as well: triangulated walls can be stored as STL or VTK files, and read-in using the `readTriangleWalls` function. However, it is generally not recommended to use triangulated walls for the following reasons: Firstly, the discretisation error of triangulating surfaces can be significant, especially for surfaces with high local curvature, such as coils or helicoidal shapes, or for moving surfaces that are only separated by a narrow gap. Secondly, as you refine your triangulation, you very quickly get to a large number of triangles, which slows down the contact detection; whereas, with the *MercuryDPM* curved wall support you have just one wall.

3.2.1. Application: Industrial mixers

All of the above-mentioned triangulation problems occur when studying industrial mixers. One such example is the Nauta-style mixer shown in Fig. 6 right. In *MercuryDPM*, this mixer is composed of only four curved surfaces: two conical walls for the casing and the base, and a helical screw with a cylindrical shaft, which rotate around their axis as well as along the casing. The high curvature of the helical screw and the narrow gap between the screw and the outer casing are hard to resolve using triangulated surfaces. Thus, triangulated geometries need to be highly refined, with thousands of triangles representing a single surface, which is less efficient and less accurate than using curved walls.

3.2.2. Application: Tunnel boring machine

Thanks to *MercuryDPM*'s support of curved walls, it was possible to simulate a Tunnel Boring Machine (TBM). TBMs are used to excavate tunnels with a circular cross section; they have a rotating wheel, called cutter-head, used to excavate the soil. When the ground is soft, Earth Pressure Balance Machines (EPB) are used. They get this name because they use the excavated material to balance the pressure at the tunnel face and this is obtained using

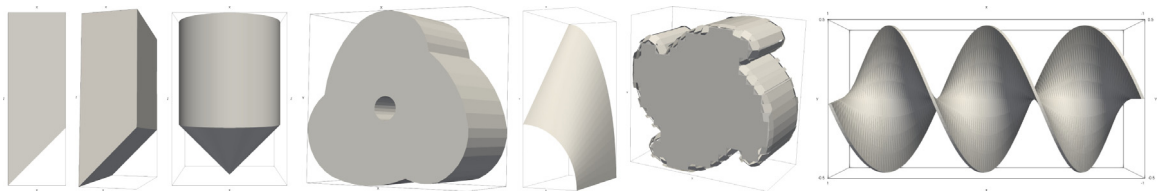

Fig. 3. Examples of the most common wall types, from left to right: polygon, polyhedron, conical shape, triangulated wall, NURBS surface; level-set surface, double-sided screw. See the source directory `Drivers/Walls/` for an implementation of the examples shown.

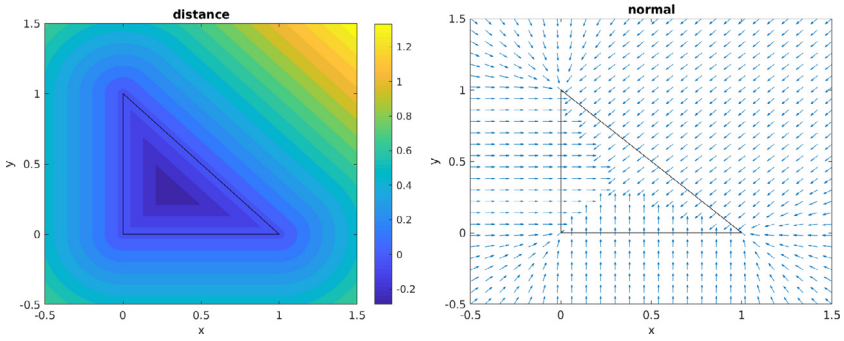

Fig. 4. Normal direction and distance computed for the contact of a particle at position (x, y) with a triangular wall spanned by the three points $(0, 0)$, $(1, 0)$, $(0, 1)$. 3-dimensional polyhedra are implemented similarly.

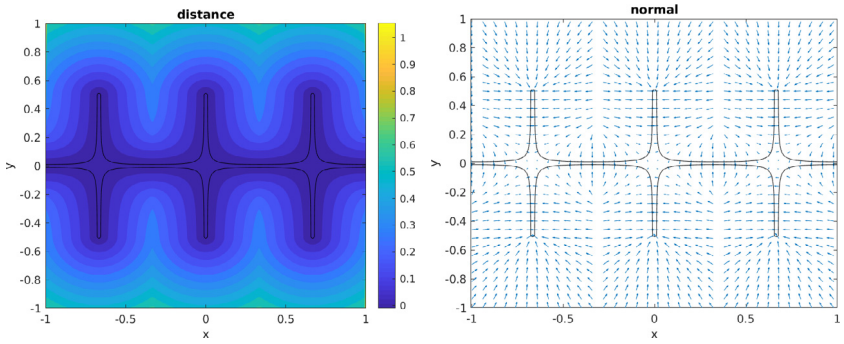

Fig. 5. Normal direction and distance computed for the contact of a particle at position $(x, y, 0)$ with a two-sided helical screw (Fig. 3 right).

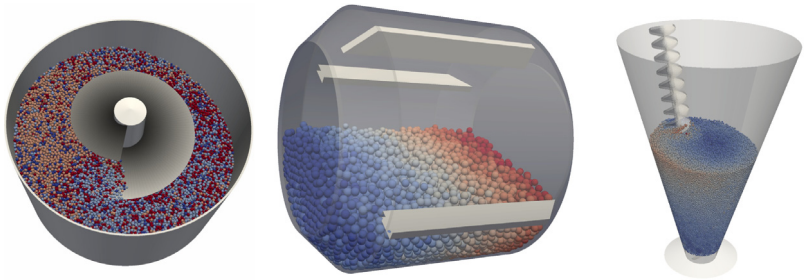

Fig. 6. Industrial mixers simulated in *MercuryDPM* with curved geometric features (no triangulation). Left-to-right: Auger mixer, rotating drum, and Nauta mixer.

a screw. The screw allows the maintenance of the prescribed pressure inside the excavation chamber. EPBs have a complex geometry, but with *MercuryDPM* it is possible to obtain a simplified version using a novel hybrid of complex and triangulated walls. A simplified EPB was obtained using already implemented shapes, like `AxisymmetricIntersectionOfWalls`, `Screw` and `TriangleWall`. The EPB's body (Fig. 7a) was created using curved shapes, while the cutter-head, which has a more complex shape, was read in as a triangulated wall from an STL file, using `readTriangleWall`. The cutter head (Fig. 7b) has been designed from a physical EPB model used in the Laboratory of Civil Engineering and Building Sciences of ENTPE in Lyon (France). With this model, it was possible to simulate the excavation phase and analyse the behaviour of the tunnelling ground on-site, varying parameters such as the EPB's velocity, the cutterhead's angular velocity and the screw's angular velocity [21].

3.3. Coarse graining

Coarse graining (CG) is a micro–macro transition method: it extracts continuum fields (density, momentum, stress, etc.) from discrete particle simulations, allowing the validation and calibration of macroscopic models [11–14]. Unlike binning methods,

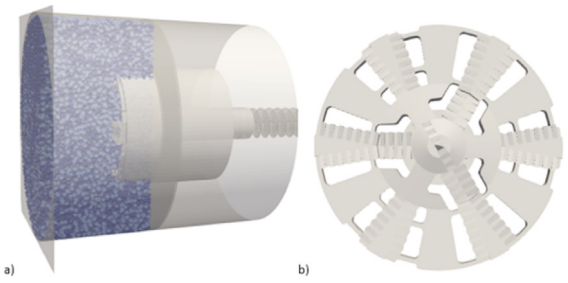

Fig. 7. (a) EPB and soil simulated with *MercuryDPM*; (b) Cutter-head created with triangulated walls [21].

which extract average values in small volumes, coarse graining evaluates continuum fields as a function of time and space. Thus, unlike binning, the resulting fields are continuous, satisfy local mass and momentum conservation exactly, and the spatial and temporal averaging scales (w and w_t) are well-defined [12,13].

The approach is flexible and the latest version can model both bulk and mixtures [13,22], boundaries and interfaces [11], non spherical particles [23], time-dependent [13], steady and static situations. It is available in *MercuryDPM* either as a post-processing tool or it can be run in real-time, i.e., concurrent with the simulation.

The aim of coarse-graining is to define continuum fields that automatically satisfy the continuum model. Most continuum models are based on mass and momentum conservation; in that case, we need to define density ρ, velocity $\mathbf{u}$ and stress $\boldsymbol{\sigma}$ such that these fields satisfy

$$\partial_t \rho + \nabla(\rho \mathbf{u}) = 0, \tag{1}$$

$$\partial_t(\rho \mathbf{u}) + \nabla(\rho \mathbf{u} \otimes \mathbf{u}) = -\nabla \boldsymbol{\sigma} - \rho \mathbf{g}. \tag{2}$$

Spatial coarse-graining defines density by applying a smoothing kernel $\phi(\mathbf{r})$ to the statistical-mechanics definition of density,

$$\rho_m(\mathbf{r}) = \sum_{i=1}^{N} m_i \delta(\mathbf{r} - \mathbf{r}_i) :$$

$$\rho(\mathbf{r}) = \rho_m \circ \phi = \sum_{i=1}^{N} m_i \phi(\mathbf{r} - \mathbf{r}_i).$$

The integral over density should equal mass, density should be non-negative, and the density at a point $\mathbf{r}$ should only depend on the particles within the neighbourhood of $\mathbf{r}$. Therefore, we require that the kernel function ϕ:

- is normalised: $\int_{\mathbb{R}^3} \phi(\mathbf{r})\, d\mathbf{r} = 1$,
- is non-negative: $\phi(\mathbf{r}) \geq 0$ for all $\mathbf{r} \in \mathbb{R}^3$,
- has compact support: $\exists c \in \mathbb{R}$: $\phi(\mathbf{r}) = 0$ for all $|\mathbf{r}| > c$.

A typical coarse-graining function is the cut-off Gaussian,

$$\phi^{\tilde{G}}(\mathbf{r}) = \begin{cases} C \exp\left(-\frac{|\mathbf{r}|^2}{2w^2}\right) & \text{if } |\mathbf{r}| < 3w, \\ 0 & \text{else,} \end{cases}$$

with appropriate prefactor C, or cut-off polynomial functions such as the Lucy kernel,

$$\phi^{L}(\mathbf{r}) = \begin{cases} \frac{105}{16\pi c^3}(-3(|\mathbf{r}|/c)^4 + 8(|\mathbf{r}|/c)^3 \\ \quad -6(|\mathbf{r}|/c)^2 + 1) & \text{if } |\mathbf{r}| < c, \\ 0 & \text{else,} \end{cases}$$

which is smoother (2nd-order differentiable) and more efficient to evaluate than $\phi^{\tilde{G}}$; see [12] for details.

To satisfy (1), velocity is defined as

$$\mathbf{u} = \frac{\mathbf{j}}{\rho}, \quad \mathbf{j}(\mathbf{r}) = \sum_{i=1}^{N} m_i \mathbf{v}_i \phi(\mathbf{r} - \mathbf{r}_i).$$

Similarly stress has to be defined to satisfy (2). Thus, $\boldsymbol{\sigma} = \boldsymbol{\sigma}^k + \boldsymbol{\sigma}^c$ with

$$\boldsymbol{\sigma}^k = \sum_{i=1}^{N} m_i \mathbf{v}_i' \mathbf{v}_i' \phi(\mathbf{r} - \mathbf{r}_i),$$

$$\boldsymbol{\sigma}^c = \sum_{i=1}^{N} \sum_{j=1}^{N} \mathbf{r}_{ij} \otimes \mathbf{f}_{ij}\ \psi_{ij}(\mathbf{r}) + \sum_{k=1}^{N_\mathrm{w}} \mathbf{r}_{ik} \otimes \mathbf{f}_{ik}^{\mathrm{w}}\ \psi_{ik}(\mathbf{r}),$$

with fluctuation velocity $\mathbf{v}_i' = \mathbf{v}_i - \mathbf{u}$, branch vector $\mathbf{r}_{ij} = \mathbf{c}_{ij} - \mathbf{r}_i$, and line integral

$$\psi_{ij}(\mathbf{r}) = |\mathbf{r}_{ij}|^{-1} \int_0^1 \phi(\mathbf{r} - \mathbf{r}_i - s\mathbf{r}_{ij})\, \mathrm{d}s.$$

Note, this integral can usually be computed exactly, without requiring numerical quadrature. For e.g. a Gaussian kernel we obtain

$$\psi_{ij}^{G}(\mathbf{r}) = \frac{1}{(2\pi w^2)^{3/2}} \exp\left(-\frac{|\mathbf{t}|^2}{2w^2}\right) \operatorname{erf}\left(\frac{n}{2w}\right)\Big|_{n=n_0}^{n_1},$$

where $n_1 = (\mathbf{r} - \mathbf{r}_i) \cdot \mathbf{n}_{ij}$, $n_0 = (\mathbf{c}_{ij} - \mathbf{r}) \cdot \mathbf{n}_{ij}$ and $\mathbf{t} = \mathbf{r} - \mathbf{r}_i - n_1 \mathbf{n}_{ij}$. For other coarse-graining functions (cutoff Gaussian, polynomials), the definitions of ψ_{ij} are more complex, but also explicit.

Output data can be coarse-grained in *MercuryDPM* using the `MercuryCG` tool.[1] For example, the following command will apply CG to the output of the `FiveParticles` application:

```
MercuryCG FiveParticles -stattype XZ -n 200 -w 0.1 -tmin 20
```

Because this simulation is two-dimensional, we resolve spatially in x and z only (`-stattype XZ`), on a grid of 200 × 200 points (`-n 200`), using a spatial coarse-graining width $w = 0.1$ (`-w 0.1`). Only the last time step $t = 20$ is evaluated (`-tmin 20`), where the simulation is steady. The output can be visualised in Matlab using `loadstatistics.m`:

```
>> data = loadstatistics("FiveParticles.stat");
>> contourf(data.x, data.z, data.Density);
```

The result is shown in Fig. 8 (centre). More examples can be found in [24].

MercuryDPM is the only code where coarse-graining can be applied during a simulation. This allows for efficient computation of high-resolution continuum fields without the need for large output files. It further allows for a two-way coupling between the continuum fields and the particle simulation, e.g. for solid–particle coupling (force-controlled walls) and particle–fluid coupling (suspensions); see Section 9 for details.

`MercuryCG` can also be applied to data from other DPM simulation softwares, and even experiments: it has e.g. been used to analyse experimental data from particle position tracking [25].

There is a lot more to say on the details of coarse-graining, and we will soon publish more details about the algorithm.

Note, the `MercuryCG` tool is currently being updated, and will soon get a new interface, and more capabilities, including: temporal smoothing kernels; coarse-grained liquid distribution and coarse-grained particle size distribution; Lagrangian-style coarse-graining where the evaluation points move with the flow [26]; and the ability to define your own coarse-grained fields.

[1] The tool was called `fstatistics` in previous publications.

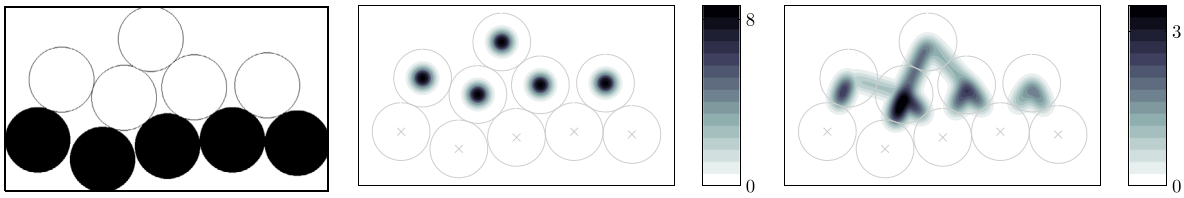

Fig. 8. Snapshot of the final state of the FiveParticles simulation (left). Coarse-graining is applied to obtain the bulk density ρ (centre) and pressure p (right).

4. Boundary conditions

Basic particle simulations start with an assembly of particles and walls, and simulate the particle (and wall) motion over time. However, many process simulations require more complex setups, including the insertion or deletion of particles during the simulations; periodic boundary conditions; or stress-, displacement- or temperature-control. Such extensions to the basic DPM simulation setup have the suffix `Boundary` in *MercuryDPM*, and are stored in the `BoundaryHandler`. We now review the most commonly used boundaries.

4.1. Insertion boundaries

Insertion boundaries are used to insert new particles during the simulation. Most commonly used is the `CubeInsertionBoundary`, which inserts particles in a rectangular region of the domain. The user can define the insertion region, a (variable or constant) insertion rate; a particle size distribution; and a velocity distribution. Other insertion boundaries have been implemented for different geometries of the insertion region (`ChuteInsertionBoundary`, `HopperInsertionBoundary`).

4.2. Deletion boundaries

Deletion boundaries are used to remove particles during the simulation. The `CubeDeletionBoundary` removes all particles that enter a rectangular section of the domain.

4.3. Periodic boundaries

Periodic boundary conditions are used to simulate small representative volume elements, allowing the study of certain flow or deformation conditions without the influence of walls. Uni-, bi- or triaxial compression, simple shear flow, chute flow, Hele-Shaw or Couette geometries are just some of the many examples. *MercuryDPM* has three kinds of periodic boundary conditions:

- `PeriodicBoundary` simulates a periodic region between two parallel planes.
- `AngledPeriodicBoundary` simulates a wedge-shaped periodic region between two non-parallel planes
- `LeesEdwardsBoundary` consists of two periodic boundary in x- and y-direction. Particles crossing the y-boundary experience a shift in x-velocity $\Delta v_x(t)$ and a shift in x-position $\Delta x = \int v_x \,\mathrm{d}t$. This forces a shear velocity profile in steady-state.

The implementation uses ghost particles, i.e. copies of real particles that are close to the periodic boundary, in order to transfer forces across the periodic boundary. A sketch of the three boundary conditions is given in Fig. 9.

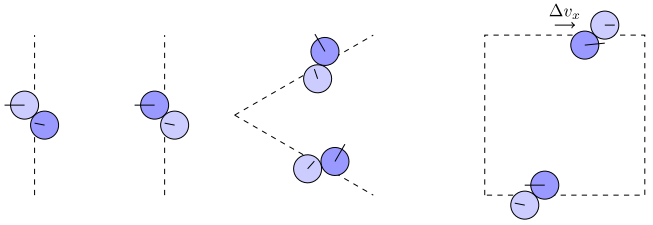

Fig. 9. Three different periodic boundary conditions, from left to right: `PeriodicBoundary`, `AngledPeriodicBoundary`, `LeesEdwardsBoundary`. The boundaries are shown as dashed lines. In each case, a particle pair is shown at the edge of the boundary. Dark blue particles are actual particles, light blue particles are ghost particles. Solid lines indicate particle velocity.

4.4. Stress- and strain-controlled periodic boundaries

The `StressStrainControlBoundary` simulates a wide range of stress- and strain-controlled shear flow and compression tests [27]. It combines a normal periodic boundary in z-direction with a Lees–Edwards boundary in x and y [28–30], as shown in Fig. 10. The user can specify a combination of targets for the stress and strain rate tensor. The advantage of this boundary is the freedom of choosing the control parameters freely, allowing the user to specify both a target stress tensor, $\boldsymbol{\sigma}$, and a strain-rate tensor, $\dot{\boldsymbol{\epsilon}}$, as input parameters. For example,

- for constant-stress uniaxial compression, specify $\boldsymbol{\sigma} = (\sigma_{xx}\ 0\ 0,\ 0\ 0\ 0,\ 0\ 0\ 0)$ and set $\dot{\boldsymbol{\epsilon}}$ to zero;
- for constant-rate uniaxial compression, set $\boldsymbol{\sigma}$ to zero and specify $\dot{\boldsymbol{\epsilon}} = (\dot{\epsilon}_{xx}\ 0\ 0,\ 0\ 0\ 0,\ 0\ 0\ 0)$;
- for triaxial compression, which is mostly used in sample preparation to achieve a homogeneous initial packing, set $\boldsymbol{\sigma} = (\sigma_{xx}\ 0\ 0,\ 0\ \sigma_{yy}\ 0,\ 0\ 0\ \sigma_{zz})$ and $\dot{\boldsymbol{\epsilon}} = 0$ for stress-control, or $\dot{\boldsymbol{\epsilon}} = (\dot{\epsilon}_{xx}\ 0\ 0,\ 0\ \dot{\epsilon}_{yy}\ 0,\ 0\ 0\ \dot{\epsilon}_{zz})$ and $\dot{\boldsymbol{\sigma}} = 0$ for volume-control.
- for constant-volume simple-shear deformation, specify $\boldsymbol{\epsilon} = (0\ \dot{\epsilon}_{xy}\ 0,\ 0\ 0\ 0,\ 0\ 0\ 0)$ and set $\boldsymbol{\sigma}$ to zero.
- for constant-stress simple-shear deformation, specify $\boldsymbol{\sigma} = (\sigma_{xx}\ 0\ 0,\ 0\ \sigma_{yy}\ 0,\ 0\ 0\ \sigma_{zz})$ and $\boldsymbol{\epsilon} = (0\ \dot{\epsilon}_{xy}\ 0,\ 0\ 0\ 0,\ 0\ 0\ 0)$, to have the stress adapt while shearing at a constant rate.

Note that the same element in the target stress and strain-rate tensors cannot be set simultaneously, e.g. the user could not set $\boldsymbol{\sigma} = (\sigma_{xx}\ 0\ 0,\ 0\ 0\ 0,\ 0\ 0\ 0)$ with $\dot{\boldsymbol{\epsilon}} = (\dot{\epsilon}_{xx}\ 0\ 0,\ 0\ 0\ 0,\ 0\ 0\ 0)$ at the same time, or the two control targets will conflict with each other, resulting in an invalid deformation mode. Further, because the Lees–Edwards boundary conditions are applied in the xy-plane, shear can only be applied in the xy-direction (not in xz and yz). However, one can achieve all possible physical constant-rate deformation modes with 3 diagonal elements and one off-diagonal element.

4.5. Other boundary conditions

Many other interesting boundary conditions have been implemented in *MercuryDPM*:

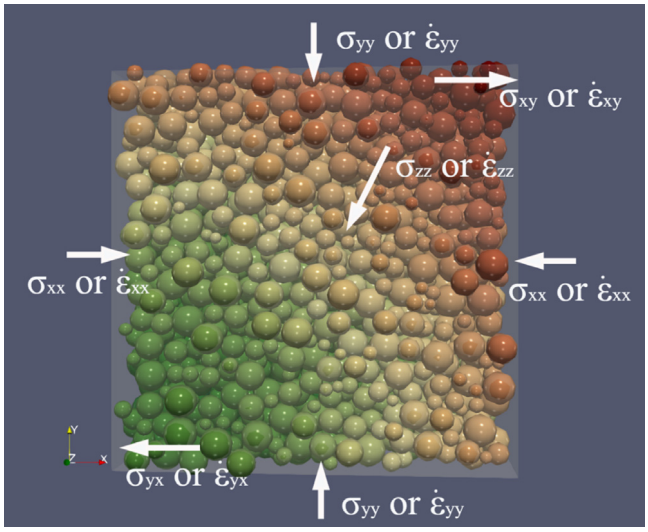

Fig. 10. Stress–strain controlled periodic boundary condition in *MercuryDPM* and its possible deformation modes on a representative element volume (REV).

Table 1
Contact forces implemented in *MercuryDPM*.

Normal forces: `Name`
Linear spring-dashpot [32]: `LinearViscoelastic`
Hertz spring-dashpot [33]: `HertzianViscoelastic`
Linear elasto-plastic cohesive [32]: `LinearPlasticViscoelastic`
Solid-state sintering [34]: `Sinter`
Melting particle model [35]: `Meltable`
Frictional forces and torques: `Name`
Sliding friction
- for linear normal force [32]: `SlidingFriction`
- for Hertz normal force [33]: `Mindlin`
Sliding, rolling, and torsion friction
- for linear normal force [32]: `Friction`
- for Hertz normal force [33]: `MindlinRollingTorsion`
Adhesive/short-range forces: `Name`
Reversible linear adhesion [36]: `ReversibleAdhesive`
Irreversible linear adhesion [36]: `IrreversibleAdhesive`
Liquid bridge adhesion [36]: `LiquidBridgeWillet`
Migrating liquid bridges [37]: `LiquidMigrationWillet`
Permanent particle bonds [38]: `Bonded`
Charged particles [38]: `ChargedBonded`

- `SubcriticalMaserBoundary` and `ConstantMassFlowMaserBoundary` insert particles from a periodic system into a larger setup, such as chute flows [31], allowing the simulation of steady-state inflow conditions.
- `HeaterBoundary` acts similar to a thermostat, supplying a heat flux to a specific region;
- `FluxBoundary` does not affect the simulation but measures flow rates through a given plane.

We refer the reader to the documentation (`docs.mercurydpm.org`) for further reading.

5. Contact models

Contact models are used to determine the forces acting between particle pairs. Many different contact forces have been described in literature, which can roughly be classified into three categories: elastic, plastic and dissipative forces f_{ij}^{n} that act in the normal direction to the contact area, $\mathbf{n}_{ij}$; tangential forces $\mathbf{f}_{ij}^{\mathrm{t}}$ and torques $\boldsymbol{\tau}_{ij}$ due to sliding, rolling and torsion friction; and adhesive normal forces f_{ij}^{a} that may act between nearby particles even if they are not in contact. Which contact model best describes the real contact behaviour depends on the material type and particle size, and on ambient effects such as temperature and moisture. In most cases, a combination of these forces needs to be taken into account, i.e. the total contact force is given as

$$\mathbf{f}_{ij} = (f_{ij}^{\mathrm{n}} + f_{ij}^{\mathrm{a}})\mathbf{n}_{ij} + \mathbf{f}_{ij}^{\mathrm{t}}.$$

All contact models in *MercuryDPM* are defined by combining a normal, frictional, and adhesive contact model. The normal, frictional and adhesive contact models currently available in *MercuryDPM* are summarised in Table 1. The name of a contact model is obtained by concatenating the names of the normal, frictional, and adhesive contact model and adding the word `Species`. For example, particles of type `LinearViscoelasticFrictionLiquidBridgeWilletSpecies` interact with a linear spring-dashpot normal force, sliding, torsion and rolling friction, and liquid-bridge adhesion forces. All contact models require a normal force, but frictional and adhesive forces are optional. Thus, `LinearViscoelasticSpecies` denotes the simple linear spring-dashpot contact model.

5.1. Normal force models

For a pair of spherical particles i, j with radii a_i, a_j and position $\mathbf{r}_i$, $\mathbf{r}_j$, we first compute the distance vector $\mathbf{r}_{ij} = \mathbf{r}_j - \mathbf{r}_i$, then the overlap $\delta_{ij}^{\mathrm{n}} = a_1 + a_2 - |\mathbf{r}_{ij}|$, the unit normal $\mathbf{n}_{ij} = \mathbf{r}_{ij}/|\mathbf{r}_{ij}|$, the branch vector $\mathbf{r}_{ij} = (a_i - \delta_i/2)\mathbf{n}_{ij}$ and the contact point $\mathbf{c}_{ij} = \mathbf{r}_i + \mathbf{r}_{ij}$. The same quantities need to be defined for pairs of non-spherical particles and particle–wall contacts, as these quantities are necessary to define the contact force.

We further define the relative velocity at the contact point, $\mathbf{v}_{ij} = (\mathbf{v}_i + \mathbf{r}_{ij} \times \boldsymbol{\omega}_i) - (\mathbf{v}_j + \mathbf{r}_{ji} \times \boldsymbol{\omega}_j)$, and its normal component, $v_{ij}^n = \mathbf{v}_{ij} \cdot \mathbf{n}_{ij}$.

The normal contact force f_{ij}^n is nonzero if the two particles are in contact, i.e. if their overlap is positive. Several different normal force models are implemented:

5.1.1. Linear spring-dashpot model

The linear spring-dashpot model, implemented as `LinearViscoelasticSpecies`, is defined as

$$f_{ij}^{\mathrm{n}} = \begin{cases} k_{\mathrm{n}}\delta_{ij}^{\mathrm{n}} + \gamma_{\mathrm{n}} v_{ij}^n & \text{if } \delta_{ij}^n > 0, \\ 0 & \text{else,} \end{cases} \tag{3}$$

with stiffness (or spring constant) $k_{\mathrm{n}} > 0$ and damping coefficient $\gamma_{\mathrm{n}} \geq 0$. The force–displacement relation is shown in Fig. 11 left. The model is efficient and simple to analyse, with collision time t_c and restitution coefficient ϵ_{n} only dependent on particle mass, not relative velocity. It is an appropriate contact models for large particles (>100 μm) or upscaled systems, where one particle represents a conglomerate of particles. For large deformation, plasticity needs to be taken into account, see Section 5.2. For sound and compaction experiments, calibrating the stiffness is important. For flows, stiffness just has to be sufficiently high such that the deformations remain small.

5.1.2. Hertz spring-dashpot model

The Hertz elastic normal force (`HertzianViscoelasticSpecies`) is based on the contact between perfectly elastic, spherical particles [39]. It is based on measurable material parameters (the elastic modulus and Poisson ratios) and accurate for powders and small deformations of larger granules, as they appear e.g. in sound propagation experiments. In most other experiments, however, the choice of stiffness is either of little importance, as other effects (such as dissipation, friction or plasticity) become dominant.

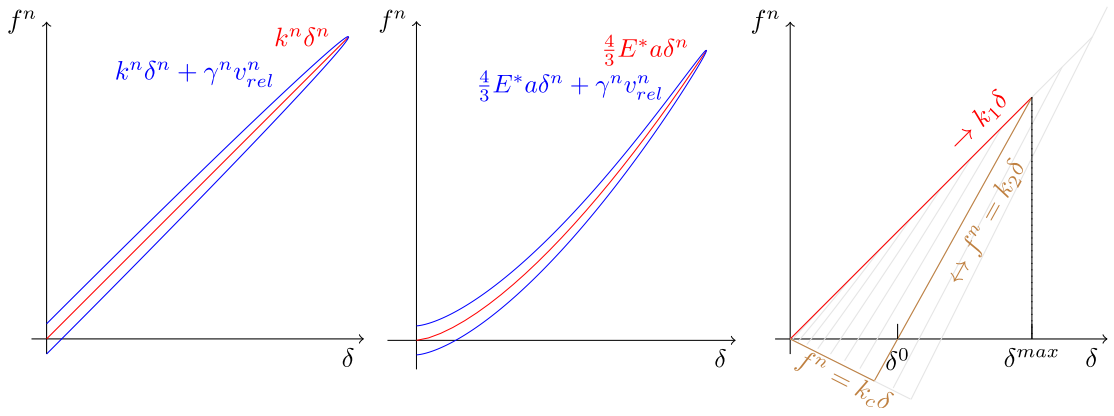

Fig. 11. Linear (left) and Hertz (middle) elastic force model, without (red) and with (blue) dissipation. Right: Plastic-adhesive force (without dissipation) for intermediate loading. Red denotes the loading, brown the unloading branch.

The Hertz force model is given by (3), but the stiffness k^n is now a variable of the radius of the contact area a_{ij}^c,

$$k^n = \frac{4}{3}E_{ij}^{\text{eff}}a_{ij}^c, \tag{4}$$

with the effective modulus $E_{ij}^{\text{eff}} = [\frac{(1-\nu_i^2)}{E_i} + \frac{(1-\nu_j^2)}{E_j}]^{-1}$, computed from the elastic moduli E_i, E_j and Poisson ratios ν_i, ν_j of the two particles. For small overlaps, we approximate the contact radius by $a_{ij}^c = \sqrt{a_{ij}^{\text{eff}}\delta_{ij}}$, with $a_{ij}^{\text{eff}} = \frac{a_i a_j}{a_i+a_j}$ the effective radius (i.e. the harmonic mean). See [39] for more information on calibrating the model.

The dissipation coefficient γ^n is chosen to satisfy the modeller's assumption on the restitution coefficient. See [40–42] for a review of the different dissipation models. In *MercuryDPM*, the dissipation coefficient γ^{n} is proportional to $\sqrt{m_{ij}^{\text{eff}}k^{\text{n}}}$, resulting in a constant restitution coefficient [43].

The proportionality of stiffness and contact radius has an important effect on the particle behaviour: It makes particles stiffer under pressure (see Fig. 11 middle), thus the speed of a pressure wave increases if the material is compressed. This is important in sound propagation but can usually be neglected in flow situations, where stiffness has only a minor effect. Furthermore, while Hertz models work well for spherical powders, the stiffness measured in macroscopic particles ($> 100\ \mu$m) is often relatively constant, due to plasticity, surface roughness and particle shape effects.

5.2. Linear elasto-plastic cohesive

To mimic plastic deformation, as observed in experiments [44], Walton and Braun [45] and Walton [46] introduced a so-called 'partially latching spring' model that used different normal spring stiffnesses for loading and unloading,

$$f_{ij}^{\text{n}} = \gamma_{\text{n}}v_{ij}^{\text{n}} + \begin{cases} k_1\delta_{ij}^{\text{n}} & \text{if } \delta_{ij}^{\text{n}} > \delta_{ij}^{\max}, \\ k_2(\delta_{ij}^{\text{n}} - \delta_{ij}^0) & \text{if } \delta_{ij}^{\min} < \delta_{ij}^{\text{n}} \le \delta_{ij}^{\max}, \\ -k_c\delta_{ij}^{\text{n}} & \text{if } 0 < \delta_{ij}^{\text{n}} \le \delta_{ij}^{\min}, \\ 0 & \text{else.} \end{cases} \tag{5}$$

To track the plastic deformation, the maximum overlap $\delta_{ij}^{\max}$ is stored and used to compute the plastic overlap δ_{ij}^0 and minimum-force overlap $\delta_{ij}^{\min}$,

$$\delta_{ij}^0 = \frac{k_2 - k_1}{k_2}\delta_{ij}^{\max}, \quad \delta_{ij}^{\min} = \frac{k_2}{k_2 + k_c}\delta_{ij}^0.$$

This model was extended by Luding [32] to allow for slowly changing stiffness: During loading, the stiffness increases linearly with the maximum overlap until a maximum unloading stiffness $\hat{k}_2$ is reached at a fraction ϕ of the effective radius,

$$k_2(\delta^{\max}) = \min\left(k_1 + (\hat{k}_2 - k_1)\frac{\delta^{\max}}{\phi a_{ij}^{\text{eff}}}, \hat{k}_2\right).$$

The force–displacement curve for this model (`LinearPlasticViscoelasticSpecies`), is shown in Fig. 11 right.

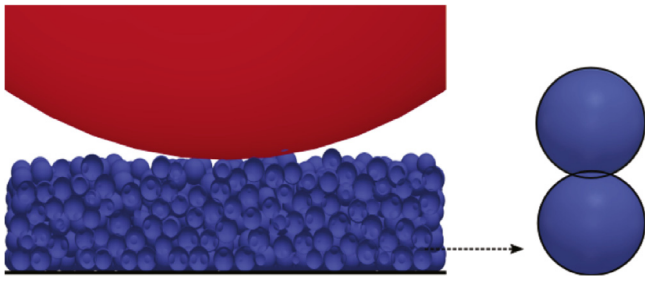

Fig. 12. Vertical cut through centre of sintered sample during indentation.

5.2.1. Solid-state sintering

Solid-state sintering is a thermal treatment for bonding particles into a solid structure. Particles are sintered by heating particles beyond the glass temperature, but below the melting point of a material. This process is controlled by transport and diffusion of material along the particle's surface and volume, which leads to a reduction of the particle surface area. Solid-state sintering has three stages [47]. The first stage is neck formation: Matter from the particle is transported from regions of high chemical potential (contact region) to regions of low chemical potential (concave neck regions). In the second stage the diameters of the pores channels shrink until the pore structure changes. In the last stage isolated pores form. All stages are dominated by different transport mechanisms, and there is a strong dependence on temperature and initial particle size in the final stage of the process. The solid-state sinter model in *MercuryDPM* (`SinterSpecies`) describes the first stage, introducing a gradual increase in plastic overlap between particles at high temperatures. In [48–50], the model is applied to sintered polystyrene particles, and numerical and experimental indentation tests are executed and compared, see Fig. 12.

5.2.2. Partial melting

If particles are heated beyond their melting temperature, they start to melt. The particles first melt at the surface, where the

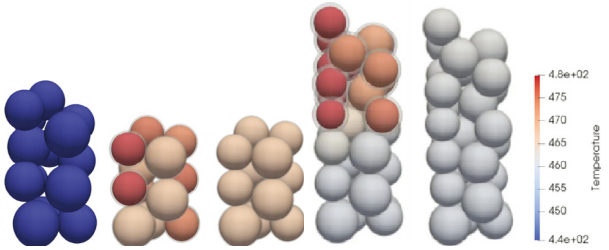

Fig. 13. Simulation snapshots, from left to right: (1) add layer of particles, (2) partial melting during heating, (3) forming bonds during cooling, (4) add second layer of particles, heating it and (5) allow it to cool down.

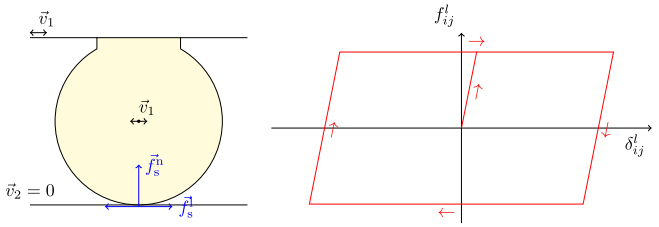

Fig. 14. Left: Schematic of a particle sliding forth and back along a surface, Right: sliding friction force vs tangential displacement, measured experimentally [53].

heat is applied, forming a melt layer that increases in thickness until the particles are fully melted. This process can be modelled in *MercuryDPM* using the `MeltableSpecies` and was initially developed for additive manufacturing processes. In particular, we consider powder bed fusion (PBF), where objects are produced by spreading successive layers of powdered material and hardening selected parts by partially or fully melting them with a laser. PBF processes are highly sensitive to the powder characteristics; therefore, the process parameters need to be optimised for each material. This is typically done by performing costly experimental trials, so developing a computational tool capable of capturing the stochastic nature of the process will help in reducing the amount of trials and thus lower manufacturing costs. In addition, particle-scale simulations of the spreading process can provide information on the powder layer behaviour and quality that is not accessible by experiments (porosity, particles segregation, etc.). A parametric study of the influence of inter-particle friction on the powder layer quality has been done by [51]. `MeltableSpecies` is based on the model of [52], which was applied to the formation and cyclic melting of faults during earthquakes. It is assumed that solid particles can melt during heating. On cooling these melt layers solidify, potentially forming permanent bonds between the particles. The model was modified and extended to include thermal conduction, radiation and convection. Further extensions will include phase transformation and laser heat input modelling. Fig. 13 shows a snapshot of particles partial melting, with particles diameter range 40–50 μm, and illustrates the heating and solidification of a new layer of particles.

5.3. Tangential force and torque models

5.3.1. Sliding friction

Similarly to the normal forces, one can define forces in tangential direction. For this, we define the lateral relative velocity,

$$\mathbf{v}_{ij}^{\mathrm{l}} = \mathbf{v}_{ij} - v^{\mathrm{n}}\mathbf{n}_{ij},$$

and the tangential elastic displacement $\boldsymbol{\delta}_{ij}^{\mathrm{l}}$, which is set to $\mathbf{0}$ at the initiation of contact, and incremented after every timestep by the formula

$$\tilde{\boldsymbol{\delta}} \leftarrow \boldsymbol{\delta}_{ij}^{\mathrm{l}} - (\boldsymbol{\delta}_{ij}^{\mathrm{l}} \cdot \mathbf{n}_{ij})\mathbf{n}_{ij},$$
$$\boldsymbol{\delta}_{ij}^{\mathrm{l}} \leftarrow (|\boldsymbol{\delta}_{ij}^{\mathrm{l}}|/|\tilde{\boldsymbol{\delta}}|)\,\tilde{\boldsymbol{\delta}} + \mathbf{v}_{ij}^{\mathrm{l}}\Delta t. \tag{6}$$

The two-step procedure is necessary to keep the tangential displacement in the tangential plane while the particle pair rotates. Note that for a fixed normal direction $\mathbf{n}_{ij}$, we obtain $\frac{d}{dt}\boldsymbol{\delta}_{ij}^{\mathrm{l}} = \mathbf{v}_{ij}^{\mathrm{l}}$.

An elastic and dissipative lateral force can then be defined as

$$\mathbf{f}_{ij}^{\mathrm{l}} = k^{\mathrm{l}}\boldsymbol{\delta}_{ij}^{v} + \gamma^{\mathrm{l}}\mathbf{v}_{ij}^{\mathrm{l}}.$$

If the lateral force exceeds a certain level, the particle begins to slide. This is modelled by a Coulomb yield criterion, cutting off the elastic displacement when it exceeds a certain fraction μ^{l} (the sliding friction coefficient) of the normal force.

$$|\mathbf{f}_{ij}^{\mathrm{l}}| \le \mu^{\mathrm{l}} f_{ij}^{\mathrm{n}}.$$

The model, shown schematically in 14, agrees well with experimental data.

The above sliding friction model is implemented in the `SlidingFrictionSpecies`, and is intended to be used with a linear normal contact force. For the Hertzian normal force, the `MindlinSpecies` is more appropriate, which has a variable tangential stiffness k_t that depends on the effective shear modulus; see [33] for details.

5.3.2. Rolling and torsion torque

Similar to sliding friction resisting lateral motion, rolling and torsion torques are modelled to resist angular motion. Like sliding friction, these torques are modelled as elastic and dissipative with a yield criterion. For this, we define the rolling and torsion velocity,

$$\mathbf{v}_{ij}^{\mathrm{ro}} = a_{ij}^{\mathrm{eff}}(\boldsymbol{\omega}_{ij} \times \mathbf{n}_{ij}), \quad \mathbf{v}_{ij}^{\mathrm{to}} = a_{ij}^{\mathrm{eff}}(\boldsymbol{\omega}_{ij} \cdot \mathbf{n}_{ij})\mathbf{n}_{ij}.$$

Their respective displacements $\boldsymbol{\delta}_{ij}^{\mathrm{ro}}$, $\boldsymbol{\delta}_{ij}^{\mathrm{to}}$ are defined equivalently to (6). We then define elastic-dissipative rolling and torsion torques

$$\boldsymbol{\tau}_{ij}^{\mathrm{ro}} = a_{ij}^{\mathrm{eff}}\mathbf{n}_{ij} \times \left(k^{\mathrm{ro}}\boldsymbol{\delta}_{ij}^{\mathrm{ro}} + \gamma^{\mathrm{ro}}\mathbf{v}_{ij}^{\mathrm{ro}}\right),$$
$$\boldsymbol{\tau}_{ij}^{\mathrm{to}} = a_{ij}^{\mathrm{eff}}\mathbf{n}_{ij} \cdot \left(k^{\mathrm{to}}\boldsymbol{\delta}_{ij}^{\mathrm{to}} + \gamma^{\mathrm{to}}\mathbf{v}_{ij}^{\mathrm{to}}\right)\mathbf{n}_{ij},$$

with $a_{ij}^{\mathrm{eff}} = \frac{|\mathbf{r}_{ij}|\,|\mathbf{r}_{ji}|}{|\mathbf{r}_{ij}|+|\mathbf{r}_{ji}|}$ the effective length of the branch vectors. If these torques exceed a certain fraction of the normal contact force, the particle begins to roll or torque, respectively. Thus, the elastic displacement is cut off to satisfy

$$|\boldsymbol{\tau}_{ij}^{\mathrm{ro}}| \le \mu^{\mathrm{ro}} a_{ij}^{\mathrm{eff}} f_{ij}^{\mathrm{n}}, \quad |\boldsymbol{\tau}_{ij}^{\mathrm{to}}| \le \mu^{\mathrm{to}} a_{ij}^{\mathrm{eff}} f_{ij}^{\mathrm{n}}.$$

Similar to the sliding friction model, this model (`FrictionSpecies`) is intended to be used with a linear normal contact force. For the Hertzian normal force, the `MindlinRollingTorsionSpecies` is more appropriate, see [33] for details.

5.4. Adhesive force models

5.4.1. Linear reversible adhesive force

The simplest adhesion model in *MercuryDPM*, `ReversibleAdhesiveSpecies`, models a linear elastic-dissipative short-range force,

$$f_{ij}^{\mathrm{a}} = \begin{cases} -f_{\max}^{\mathrm{a}} & \text{if } \delta_{ij}^{n} \ge 0, \\ -f_{\max}^{\mathrm{a}} - k_c\delta_{ij}^{\mathrm{n}} & \text{if } -f_{ij}^{\mathrm{a,max}}/k_c \le \delta_{ij}^{\mathrm{n}} < 0, \\ 0 & \text{else.} \end{cases}$$

It is called *reversible*, because the adhesive force is equal during loading and unloading. This model is mostly used to model dry cohesion, i.e. attractive forces due to close proximity between surfaces, such as van-der-Walls interactions.

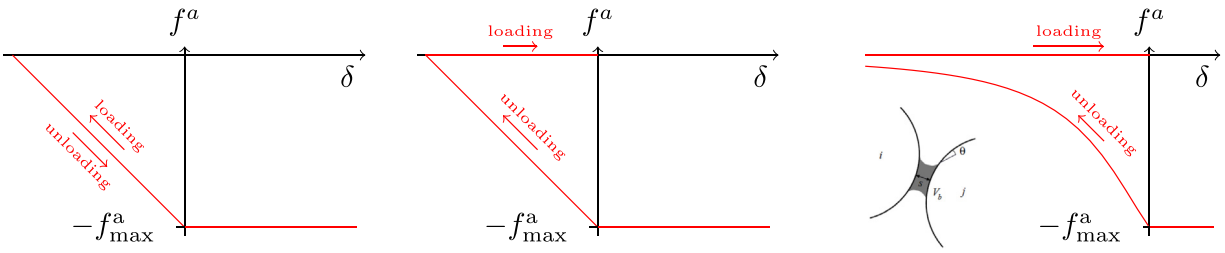

Fig. 15. The reversible linear adhesive force (*left*) follows the same curve in loading and unloading directions. The irreversible linear adhesive forces (*middle*) and the liquid bridge force model (*right*), however, have separate loading and unloading paths (illustrated by the arrows). This reflects a modification of the contact properties (e.g. snap-in of a liquid bridge, formation of chemical bonds) when the contact is established.

5.4.2. Linear irreversible adhesive force

`IrreversibleAdhesiveSpecies` is an *irreversible* linear elastic-dissipative short-range force, where the short-range adhesive force is active only during unloading (i.e. after the particles have been in contact)

$$f_{ij}^{\mathrm{a}} = \begin{cases} -f_{\max}^{\mathrm{a}} & \text{if } \delta_{ij}^{n} \geq 0, \\ -f_{\max}^{\mathrm{a}} - k_c \delta_{ij}^{\mathrm{n}} & \text{if } -f_{ij}^{\mathrm{a,max}}/k_c \leq \delta_{ij}^{\mathrm{n}} < 0 \text{ and } \delta_{ij}^{\max} > 0, \\ 0 & \text{else.} \end{cases}$$

Such models are often used to model wet cohesion, i.e. liquid bridges between particle pairs, which form when the particle pair gets in contact, but persist during unloading until the liquid bridge snaps.

Both contact models are sketched in Fig. 15.

5.4.3. Liquid-bridge cohesion

`LiquidBridgeWilletSpecies` is a nonlinear model to model liquid bridges [36,54], based on the theoretical (and experimentally validated) results of Willet et al. [55]. This force–displacement relation has been derived from first principles, based on solving the Young–Laplace equation, resulting in the following force model, shown in Fig. 15.

$$f_{ij}^{\mathrm{a}} = \begin{cases} -f_{ij}^{a,\max} & \text{if } \delta_{ij} \geq 0, \\ -\frac{f_{ij}^{a,\max}}{1+1.05\hat{\delta}_{ij}+2.5\hat{\delta}_{ij}^2} & \text{if } -S_c < \delta_{ij} < 0 \text{ and was in contact,} \\ 0 & \text{else,} \end{cases}$$

with scaled overlap $\hat{\delta}_{ij} = \delta_{ij}/(V_{\mathrm{b}}/a_{ij}^{\mathrm{eff}})^{1/2}$, rupture distance $S_c = (1+\theta/2)\sqrt[3]{V_{\mathrm{b}}}$, maximum capillary force $f_{ij}^{a,\max} = 2\pi\gamma a_{ij}^{\mathrm{eff}}\cos\theta$, liquid volume V_{b}, contact angle θ, and surface tension γ.

This model has further been extended to account for liquid migration, implemented in `LiquidMigrationWilletSpecies`. The methodology is quite straightforward: Particles and liquid are considered as two different entities in the system. Liquid is either attached to the particles (as a thin liquid film), or to the contacts (as liquid bridges). Liquid is transferred whenever contacts are formed or broken. Thus, when a contact is formed between two particles, the liquid attached to the particles can form a liquid bridge [37]. When a liquid bridge is ruptured, the bridge volume is distributed to neighbouring particles and contacts; total liquid conservation is ensured. The microscopic simulations of liquid migration has been used to validate a continuum scale model that describes the migration of liquid as shear-rate dependent diffusion [56].

5.4.4. Bonds and charges

Several other adhesive force models exist: `BondedAdhesiveSpecies` allows the user to specify a strong adhesive force f_{ij}^{b} between particles, which bonds the particles together. A bond force can be turned on or off for each individual particle pair, allowing for the simulation of soft, bendable particle clusters:

$$f_{ij}^{\mathrm{b}} = \begin{cases} f^{\mathrm{b}} & \text{if } 0 \leq \delta_{ij} \text{ and bond is active,} \\ 0 & \text{else.} \end{cases}$$

In `ChargedBondedAdhesiveSpecies`, the bond force is combined with a short-range normal force f_{ij}^{c} simulating particle charges, which can be either repulsive or adhesive, depending whether both particles have the same or opposite charge:

$$f_{ij}^{\mathrm{c}} = \pm \begin{cases} f^{\mathrm{c,\ max}} & \text{if } 0 \leq \delta_{ij}, \\ f^{\mathrm{c,\ max}} + k^{\mathrm{c}}\delta_{ij} & \text{if } \frac{f^{\mathrm{c,\ max}}}{k^{\mathrm{c}}} \leq \delta_{ij} \leq 0, \\ 0 & \text{else.} \end{cases}$$

It has been used to simulate clay particles as elongated, string-shaped particles (in 2D) or oblate particles (in 3D) with opposite charges at centre and edge of the particles [38].

5.4.5. User-defined species

Of course, the user can also define new contact models. For this, it is easiest to modify the most similar existing model. This is described in detail in the *For Developers* section of the documentation (http://docs.mercurydpm.org).

6. Non-spherical particles

As DPM studies become more complex and detailed, many users wish to use non-spherical particles in their simulations. *MercuryDPM* supports several ways to define non-spherical particle shapes, such as multi-spheres [57], superquadrics [58], agglomeration [35], and bonding [38]. We now show different non-spherical particles, using example applications.

6.1. Ellipsoidal particles

The `SuperQuadricParticle` is used to simulate particles whose surface is a superquadric, defined by the shape-function

$$f(\mathbf{x}) := \left(|x/a|^{n_2} + |y/b|^{n_2}\right)^{n_1/n_2} + |z/c|^{n_1} = 0.$$

The parameters a, b, c determine the particle size in the x, y, z direction, and n_1, n_2 determine the roundness of the edges. For example, we get ellipsoids for $n_1 = n_2 = 2$, a cylinder with rounded edges for $n_1 \gg 2$, $n_2 = 2$, and a cuboid with rounded-edges when $n_1, n_2 \gg 2$. Contact detection and computation of the overlap is implemented similar to [58]: each particle fits into a bounding sphere of radius $r = \max(a, b, c)$; based on that radius, the particles are inserted into the hierarchical grid. Whenever the hierarchical grid finds a potential contact, it is tested whether the bounding spheres of the particles i, j intersect. If that is the case, the contact-point $\mathbf{c}_{ij}$ is the defined as the $\mathbf{x}$-value minimising the function $f_i(\mathbf{x}) + f_j(\mathbf{x})$ under the condition $f_i(\mathbf{x}) = f_j(\mathbf{x})$, where $f_i(\mathbf{x})$ is the shape-function of the particle i, translated and rotated by

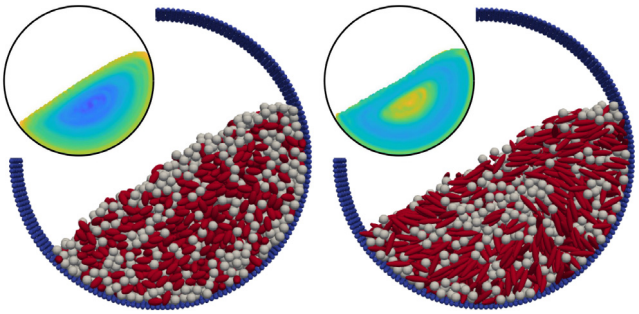

Fig. 16. Mixtures of spheres and ellipsoidal particles in a rotating drum, screenshots and coarse-grained solid volume fraction of spheres. (left) ellipsoids of aspect ratio 2, (right) ellipsoids of aspect ratio 4.

the particle's position and orientation. The particles are in contact if and only if $\mathbf{c}_{ij}$ is in the interior of both particles, i.e., $f_i(\mathbf{c}_{ij}) < 0$. From there, the direction of the contact and its overlap can be computed. For implementation details, see [59].

Note that, since the coarse-graining tool MercuryCG does not rely on particle shape, the continuum fields for these quantities can automatically be computed without any changes to the code.

To study the influence of particle elongation on segregation, we construct a rotating cylindrical drum made out of small particles. The drum is filled with mixtures of spheres and prolate ellipsoids of equal volume and equal density. After ten rotations, the mixture is coarse-grained over one and a half a rotation period in order to obtain the concentration of spheres throughout the drum. We confirmed the observation of [60] that for a combination of spheres and prolate ellipsoids with aspect ratio 2, the ellipsoids segregate to the core, while for a combination of spheres and prolate ellipsoids with aspect ratio 4, the ellipsoids segregate to the periphery of the flow; more detailed observations will be presented in a follow-up publication. Fig. 16 shows the segregation profile for both these cases.

6.2. Multispheres

For some applications, the geometry of the particles is relevant (e.g., railway ballast), and assuming spheres or ellipsoids may be a very simplistic approximation. For this reason, the concept of "multispheres" is being implemented in *MercuryDPM*. A multisphere is a cluster of spheres (or superquadrics) whose relative positions are fixed and are placed such that the boundary of the cluster approximates the intended geometry. The mass and inertia of the particle is computed such that it matches the particle's geometry. In this way, a multisphere is similar to an agglomerate of elementary particles, but no internal deformation and/or breakage is allowed. The elementary particles (slaves) composing a multisphere can overlap and their radius may vary. Due to the possible overlap of slaves composing a multisphere particle, the inertia of the multisphere cannot be calculated internally by the software; instead, it must be specified by the user.

The steps to define a multisphere particle are:

- Create a new particle, e.g. `SphericalParticle p;`
- Define its initial position (centre of gravity): `p.setPosition(Vec3D pos);`
- Define its principal axes: `p.setPrincipalDirections(Matrix3D dir);` Each column of the 3D matrix represents a principal axis; the software enforces orthogonality internally
- Define mass and inertia: `p.setMass(double mass);` `p0.setInertia(MatrixSymmetric3D inertia);`

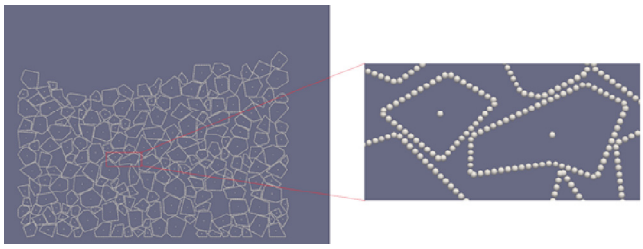

Fig. 17. Multispheres reproducing the shape of ballast particles in a simulated compression test.

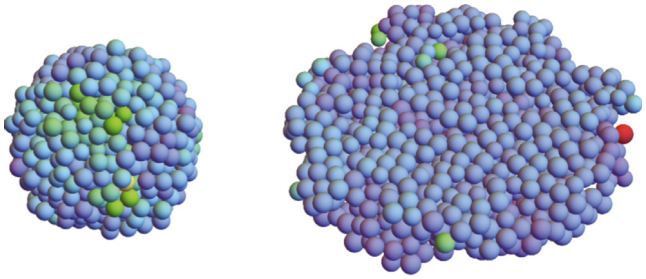

Fig. 18. Cluster composed of 1000 monodispersed particles before (left) and after (right) uniaxial compression. Color denotes kinetic energy, from blue (lowest) to green (medium) to red (highest). Most particles have very low kinetic energy, indicating a stable cluster.

- Add as many slaves as needed to achieve the intended geometry: `p.addSlave(Vec3D pos, double radius);` The position of slaves is defined relative to the centre of gravity and in terms of principal axes.

Contact forces between slave particles of a multisphere and other bodies (not belonging to the same multisphere) are calculated the same way as for any other particle, but the resulting forces and torques are applied to the multisphere's centre-of-mass. The response of the multisphere is ultimately determined by solving the equations of motion of a rigid body [57] for its (linear and angular) accelerations.

Fig. 17 depicts the application of multispheres to simulate a compression test of railway ballast material (in 2D). As can be seen, each particle is composed by small spheres delimiting the desired geometry.

6.3. Deformable/breakable clusters (agglomerates)

This new feature of *MercuryDPM* allows the user to create agglomerates (or clusters) composed of individual elementary particles. Clusters are formed by radial isotropic compression, which causes the cluster particles to adhere to one another, as shown in Fig. 18, but their relative position is not fixed, making the agglomerates deformable and breakable. This feature is useful in simulations where such properties are required, such as particle breakage [61], tableting [62], granulation, simulation of clay particles [63], etc.

The `LinearPlasticViscoelasticSpecies` [32] allows particles to be in mechanical equilibrium despite having a finite overlap, and a proportional finite tensile force is needed to pull them apart. The latter is what keeps agglomerates together, but also allows them to be deformed and broken when sufficiently strong external forces are exerted. As displayed in Fig. 18, clusters are mechanically stable before (*left*) and after (*right*) deformation. Cluster radius R and mass fraction ζ follow an analytical relation dependent on the number N of elementary particles and their plasticity ϕ [32].

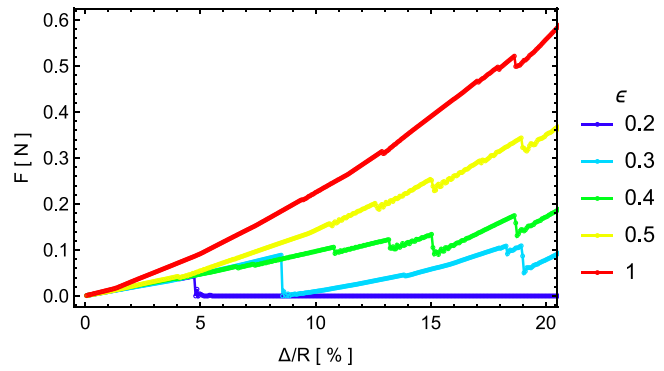

Fig. 19. Compressive force as a function of relative compression during uniaxial breakage test of a cluster with $N = 500$ and $\phi = 0.2$ for different values of ϵ. Vertical drops are due to brittle fractures of the cluster.

Since Eq. (5) is continuous, the rearrangement of particles inside of clusters due to an external (e.g. uniaxial compressive) force will be steady and gradual. This behaviour characterises plastic materials such as clay and rubber, and is only suited to model such deformation processes. However, to model brittle breakage, the elastic energy release must be sudden and trigger a fracture propagation along the body. To recover a similar behaviour the cohesive branch of the force in (5) must be discontinuous, and for this modelling purposes is substituted by

$$f_{ij}^n = \begin{cases} -k_2(\delta_{ij}^{\mathrm{n}} - \delta_{ij}^0) & \text{if } \delta_{ij}^{\mathrm{crit}} < \delta_{ij}^{\mathrm{n}} \le \delta_{ij}^0 \text{ (plasticity)}, \\ 0 & \text{if } 0 < \delta_{ij}^{\mathrm{n}} \le \delta_{ij}^{\mathrm{crit}} \text{ (breakage)}, \end{cases} \tag{7}$$

where $\delta_{ij}^{\mathrm{crit}} = \delta_{ij}^0 + \epsilon\left(\delta_{ij}^{\min} - \delta_{ij}^0\right)$ is a critical minimum overlap below which the cohesive bond is broken and ϵ is a new dimensionless material parameter. How this parameter affects the breakage behaviour is depicted in Fig. 19 where clusters are tested during uniaxial compression. Full details of this model are being prepared for a separate publication.

7. Writing applications

To write a *MercuryDPM* simulation, the class `Mercury3D` is used: this class contains the algorithm for time integration, contact detection, etc, and containers to store the elementary objects such as particles, walls, boundaries, contact models, etc. To make a new process simulation, the user creates a source file ('driver code'). In this file, one defines an (empty) object of type `Mercury3D`, and defines all the elementary objects that define the process he wants to simulate. One then calls the member function `solve()`, which calls `setupInitialConditions()` and continues the simulation. A typical user code is shown below:

```
#include "Mercury3D.h"

class Demo : public Mercury3D {
  void setupInitialConditions() override {
    //define geometric setup here
  }
};

int main() {
  Demo problem;
  //define process parameters here
  problem.solve();
}
```

The distinction between process parameters (contact law, time step, final time, domain size, etc.) and geometric setup (walls, boundary conditions, particle positions) is due to our parallelisation strategy, which requires process parameters to be set first.

7.1. General interfaces

To store elementary objects, such as particles, walls and boundary conditions, *MercuryDPM* uses a series of handler classes. The `ParticleHandler`, for example, stores all types of particles (spherical, superquadric, etc.), the `WallHandler` all types of walls, etc. This is shown in the top left of Fig. 20.

All objects in a handler share a common base class. This ensures that the syntax for all objects and handlers is the same. For example, `BaseParticle` contains the common properties of all particles, such as position, orientation, and velocity; the same member function, `getObject(int)`, can used to access an object in the `Particle`-, `Wall`-, or `BoundaryHandler`; and the same function, `getID()`, is used to access the unique id of any particle, wall, or boundary. This is shown in the bottom left of Fig. 20.

Using the inheritance structure, the user can easily define new classes of elementary objects: For example, to define a sinusoidally-shaped wall, the user creates a new class `SineWall`, inherited from `BaseWall`, introduces parameters such as amplitude and oscillation frequency, and defines the member functions, such as the `getDistanceAndNormal` function.

7.2. Code samples

Simple parameters can be defined using set-functions; a typical user code specifies the following process parameters

```
...
int main() {
  MyDriver problem;
  problem.setName("Demo");
  problem.setMin(Vec3D(-1.0,-1.0,-1.0));
  problem.setMax(Vec3D(-1.0,-1.0,-1.0));
  problem.setTimeStep(1e-4);
  problem.setTimeMax(2.0);
  problem.setGravity(Vec3D(0,0,-9.81));
  problem.setSaveCount(200);
  problem.solve();
}
```

To set an elementary object, declare it, set its parameters and add it to the appropriate handler. Below is the code to define a material of type `LinearViscoelasticSpecies`:

```
#include "Species/LinearViscoelasticSpecies.h"
...
int main(){
  ...
  LinearViscoelasticSpecies species;
  species.setDensity(2000);
  species.setStiffness(1e3);
  species.setDissipation(0.1);
  problem.speciesHandler.copyAndAddObject(species);
  ...
}
```

Now we can define geometric objects, as shown below:

```
#include "Particles/SphericalParticle.h"

class MyDriver: public Mercury3D{
  public:
  void setupInitialConditions(){
    SphericalParticle p;
    p.setSpecies(speciesHandler.getObject(0));
    p.setRadius(1e-3);
    p.setPosition(Vec3D(0.1,0.1,0.0));
    particleHandler.copyAndAddObject(p);
    ...
  }
};
...
```

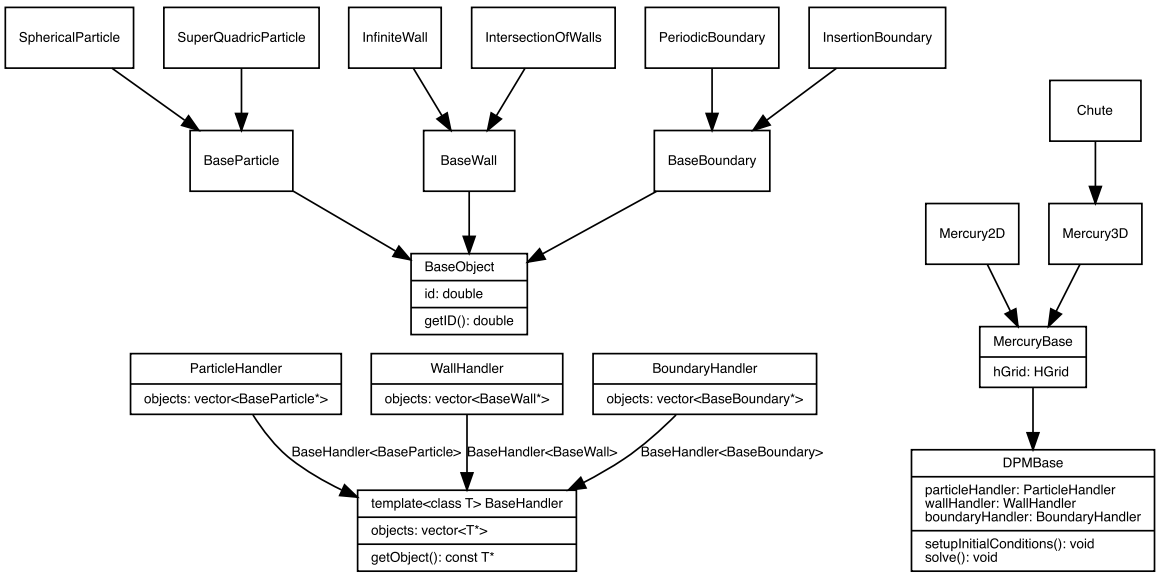

Fig. 20. Basic class structure of *MercuryDPM*, showing the inheritance and encapsulation strategy. Handlers for contact laws, interactions, coarse-graining, and MPI decomposition are not shown, and only a select number of particle, wall, and boundary types are shown.

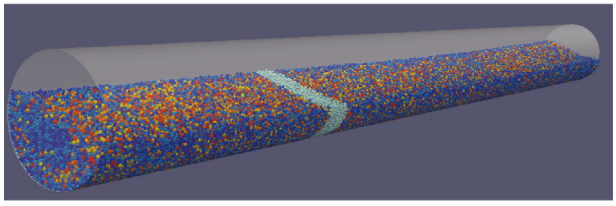

Fig. 21. A parallel simulation of a rotating drum in *MercuryDPM* using 36 cores. The light blue particles are computed on a single core.
Source: Taken from [64].

7.3. Common geometries

For most processes, the user has to define the full geometric setup from scratch. However, certain setups are so common that we have predefined them, using classes derived from Mercury3D (see right of Fig. 20). The `Chute` class, for example, contains a function to create an inclined plane, which can be rough or smooth, and has predefined periodic boundaries that allows the user to quickly setup a periodic chute flow simulation. Similarly, `ChuteWithHopper` can be used to simulate a chute with an inflow hopper. We recommend users to define their own classes with predefined setups, e.g. for parameter studies where simulations only vary slightly, and thus avoid code duplication. An application of the `Chute` class is shown in [35].

7.4. Restarting

Each `Mercury3D` class has a `write` function, which stores the current state of a simulation in a text file; and a `read` function, which reloads the written state of a simulation. This allows simulations to be restarted. This functionality can be executed via a command-line interface: for example, by calling the executable `HourGlass2DDemo`, a simulation of two seconds is launched. One can now restart this simulation and run it for a further two seconds by executing the command `HourGlass2DDemo -r HourGlass2DDemo.restart -timeMax 4`.

7.5. Parallelisation

Although *MercuryDPM* performs well for DPM simulations, due to advanced contact detecting and clever treatment of the walls, the computational power of a single processor (or thread) is limited. Thus, sequential DPM computations are limited to at most a few million particles and a few minutes of process time. In order to finish the simulation in a reasonable time, for larger computations, parallel processing is required. Currently *MercuryDPM* uses a domain-decomposition based parallel computing algorithm utilising MPI [64]. The current domain decomposition is simple: the process domain is decomposed into n_x-by-n_y-by-n_z sub-domains of equal size (as specified by the user), and each processor computes the movement of particles in one sub-domain. To determine the contacts with particles from neighbouring subdomains, a communication zone is established in the vicinity of the sub-domain boundaries, in which the processors communicate via MPI the location of the particles to their neighbours. This parallel computing algorithm can handle complex boundaries such as periodic boundaries, insertion/deletion boundaries and maser boundaries [4].

The performance of the parallel algorithm has been tested for the case of a rotating drum of varying width; a snapshot of the simulations is shown in Fig. 21. The serial algorithm shows a near-linear scaling of computing time with the number of particles (Fig. 22a). Weak scaling of the parallel implementation is shown by measuring the efficiency $E = T_p/T_s$, the ratio of computing time for the parallel and serial implementation, for a varying number of cores N_c, keeping the number of particles per core constant. On a single node, efficiency decreases slowly with the number of cores, but levels off at around 40% for simulations that use hyperthreading (Fig. 22b). On multiple nodes, hyperthreading can be avoided, and the efficiency remains above 60% (Fig. 22c). Thus, the algorithm performs very efficient for large simulations, if the computational load per core is homogeneous.

7.6. Visualisation

There are two programs to visualise *MercuryDPM* output: xBalls and ParaView. xBalls, written by Stefan Luding, is a simple X11-based viewer that allows the user to quickly check the

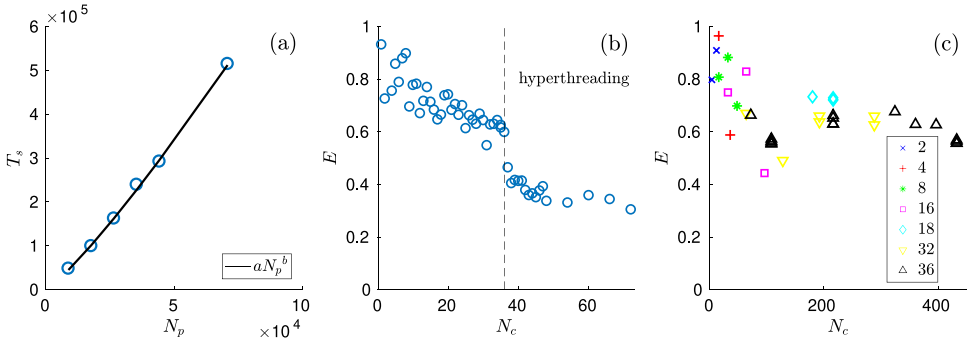

Fig. 22. (a) Computing time T_s versus particle number N_p for single-core simulations of different drum lengths ($a = 1.08$, $b = 1.17$). (b) Efficiency of the parallel algorithm on a single node with 36 physical cores. (c) Efficiency of the parallel algorithm on multiple nodes without hyper-threading. Different symbols indicate the number of cores per node.
Source: Taken from [64].

progress of the simulation. It is automatically installed with *MercuryDPM*; to visualise a simulation such as `HourGlass2DDemo` with xBalls, one simply needs to execute a script file that is part of the default simulation output, in this case `HourGlass2DDemo.xballs`. A more detailed three-dimensional visualisation of the walls and particles can be obtained via ParaView. For more information, see the *MercuryDPM* documentation at https://docs.mercurydpm.org.

8. Download, testing, documentation

8.1. Versioning

MercuryDPM is available for download at http://mercurydpm.org. One can download either the latest release, or the developer's version ("Trunk"). The Trunk is updated as soon as a new feature is complete and is intended for developers only. After six months in the Trunk (where the developers' community will be able to debug the feature), a feature is considered save to use and ready to be merged into the next release.

8.2. Self-test suite

Developing new features can have unintended consequences. For example, introducing a new variable in `DPMBase` could accidentally break the ability to restart simulations. To avoid breaking existing code by introducing new features, *MercuryDPM* uses the CTest software: Before any new code is committed to the Trunk or Release, the developer calls the command make fullTest, which (a) checks whether all codes in *MercuryDPM* compile, and (b) runs a series of self- and unit-tests to validate that no existing feature has been broken. Unit tests are designed to test a certain feature (e.g. whether the restitution coefficient is computed correctly) and return true if the test was successful; these are basic simulations that should run in less than one second. Self-tests validate more complex features (e.g. restarting), and checks whether the output files have changed; these are slightly more elaborate simulations that should run in less than 10 s. There are now more than 300 unit- and self-tests in current developer's version of *MercuryDPM*. To ensure that each feature is tested, new tests have to be committed for each new feature.

8.3. Documentation and tutorials

The documentation of *MercuryDPM* is available at docs.mercurydpm.org. All classes of *MercuryDPM* are documented here, using the Doxygen suite, which extracts documentation from comments written by the developers in the *MercuryDPM* source files. In addition, the website contains tutorials, a list of demo codes, and a basic manual that will help new users and new developers to get acquainted with *MercuryDPM*.

Finally, training materials for both C++11 (and extensions), particle simulations in general, and *MercuryDPM* is freely available from the *MercuryDPM* website.

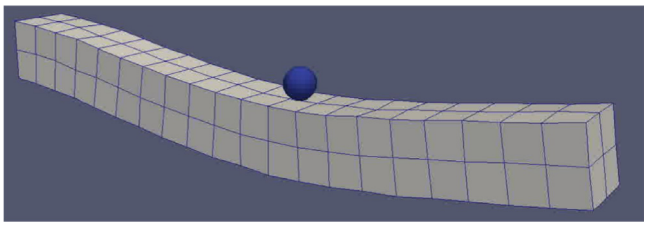

Fig. 23. Single discrete particle interacting with a solid cantilever.

9. Future direction and in-development features

9.1. Particle–solid interaction

MercuryDPM can now be coupled with oomph-lib, an object-oriented, open-source finite-element library for the simulation of multi-physics problems [65]. This allowed the development of new features, such as surface coupling with FEM walls. The coupled code can simulate walls that deform in response to the forces exerted on them by the particles, and the particle motion updated by the walls, as shown in Fig. 23.

For surface coupling, we implement wrapper classes for the solid elements (and the governing equations), provided by the oomph-lib library, for efficient, I/O-free access and assignment of the nodal/contact forces and displacements. The oomph-lib geometry is mapped onto triangulated *MercuryDPM* walls that can interact with the discrete particles. For each particle–wall interaction, the contact forces are added as external loads into the weak form of the linear momentum equation, and the wall positions and velocities in *MercuryDPM* are updated by the finite element approximation.

Applications of the coupled code include, but are not limited to: interactions between granular materials and deformable, fatigue-able structure/machinery [66], breakable discrete polygons [67], and fibre–particle mixtures [68,69].

9.2. Multi-resolution particle–fluid coupling

A second feature enabled by the coupling with oomph-lib is fluid–particle coupling. For under-resolved simulations, we use

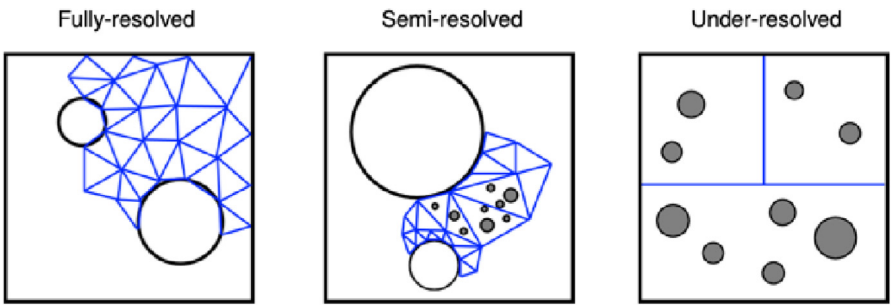

Fig. 24. Illustration of the fully-semi and under-resolved method.

the Anderson and Jackson formulation [70]. This introduces a voidage field, a measure for the fraction of total particle volume inside an element, to simulate the fluid flow. Coupling forces are be defined that specify the interaction between particles and fluid [71]. A different coupling method is semi-resolved, giving more detailed results than an under resolved method but being much faster than a fully-resolved method. The voidage can be described as continuous function by coarse-graining the particles in space. The fluid elements can then simply evaluate this function at their location. For fully resolved simulation there is a no-slip boundary condition for the fluid on the particle surface, and the coupling forces can be computed by integrating the pressure along the particle surface.

At the moment the three methods have been independent implemented; however, a multi-resolution is currently under development, where the code adapts between fully-, semi- and under-resolved situations, allowing the simulation of suspensions with arbitrary large particle size-distributions. This is illustrated in Fig. 24.

9.3. Better hybrid openMP-MPI parallelisation

The current MPI parallelisation strategy, see Section 7.5, works well for evenly distributed particle systems. However, for inhomogeneous systems, the workload of the processors is unbalanced, resulting in suboptimal scalability. This weakness of the current implementation will be addressed in the near future: one possibility to enhance load balancing is using a cyclic distribution with respect to particle identities (indices); a second possibility is to implement an adaptive mesh of differently-sized domains. These parallel-processing strategies for the *MercuryDPM* software package will be done using a combination of OpenMP and MPI for CPU parallel computing and OpenACC/CUDA for GPU computing.

9.4. STL/STEP readers for reading in industrial geometries

We have an STL reader for *MercuryDPM* but this by-passed our very nice complex curved wall support. We are currently working on an STEP reader that keeps all the curvature information, leading to quicker and more accurate simulations.

9.5. Calibration via grain learning

Calibrating contact models is a huge problem in granular materials. A wide and varied range of granular characterisation machines exists, and many of these machines produce vendor-specific, none-standard measures that are not comparable with each other. To solve this problem, MercuryDPM is integrating Grain Learning (https://github.com/chyalexcheng/grainLearning/) into its software suite. Grain learning uses Bayesian inference and machine learning to train the contact model to reproduce the supplied characterisation data [72,73]. The algorithm further identifies automatically whether the provided characterisation data is sufficient to uniquely determine the parameters of the contact model, or whether more experimentation is required. A major advantage is that the method is independent of the type of characterisation data; this makes it a very valuable tool and it will remain so until a standardisation of the measures characterising granular materials has been established, i.e., until we have granular properties that mirror surface tension, viscosity, etc. in fluid dynamics.

10. Release strategy

Originally, *MercuryDPM* has been released once a year; however, this was becoming less practical due to the large number of contributors, so we have moved to an open-development model, i.e. opening the developer's version to public download. For more information about *MercuryDPM* please visit http://MercuryDPM.org.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Thomas Weinhart: Conceptualization, Methodology, Software, Writing - original draft, Writing - review & editing, Visualization, Supervision, Project administration, Funding acquisition. **Luca Orefice:** Methodology, Software, Writing - original draft, Visualization. **Mitchel Post:** Methodology, Software, Writing - original draft, Visualization, Project administration. **Marnix P. van Schrojenstein Lantman:** Methodology, Software, Writing - original draft, Visualization. **Irana F.C. Denissen:** Methodology, Software, Writing - original draft, Visualization, Project administration. **Deepak R. Tunuguntla:** Methodology, Software, Writing - original draft, Visualization. **J.M.F. Tsang:** Methodology, Software, Writing - original draft, Visualization. **Hongyang Cheng:** Methodology, Software, Writing - original draft, Visualization. **Mohamad Yousef Shaheen:** Methodology, Software, Writing - original draft, Visualization. **Hao Shi:** Methodology, Software, Writing - original draft, Visualization. **Paolo Rapino:** Methodology, Software, Writing - original draft, Visualization. **Elena Grannonio:** Methodology, Software, Writing - original draft, Visualization. **Nunzio Losacco:** Writing - original draft, Supervision, Funding acquisition. **Joao Barbosa:** Methodology, Software, Writing - original draft, Visualization. **Lu Jing:** Methodology, Software, Writing - original draft, Visualization. **Juan E. Alvarez Naranjo:** Methodology, Software, Writing - original draft, Visualization. **Sudeshna Roy:** Methodology, Software, Writing - original draft, Visualization. **Wouter K. den Otter:** Methodology, Writing - original draft,

Funding acquisition. **Anthony R. Thornton:** Conceptualization, Methodology, Software, Writing - original draft, Writing - review & editing, Visualization, Supervision, Project administration, Funding acquisition.

Acknowledgements

The MercuryDPM team

MercuryDPM has been developed and tested by many people over the years and the code would not exist without the support of all these people. Every time we develop a few new features we try and write a short conference proceeding giving credit to both the people who developed these features and the people who tested these features [2–4,35,74].

In addition to this we try to maintain a full list of contributors on the *MercuryDPM* website, see http://mercurydpm.org/about-the-code/team.

Funding

MercuryDPM would not exist without external funding and the authors would like to acknowledge support from the following grants (since the project was started):

1. IMPACT-SIP1 "Computational multi-scale modelling of super-dispersed multiphase flows"
2. STW 11039 "Polydispersed Granular Flows through Inclined Channels"
3. NWO VICI 10828 "Bridging the gap between particulate systems and continuum theory";
4. DFG LU 450/10 "Sintering: modelling of pressure-, temperature-, or time-dependent contacts"(part of SPP 1482 "Partikel im Kontakt")
5. FOM 07PGM27 "Clustering phase diagram with dissipation and long range forces";
6. STW MuST 10120 "Molecular dynamics simulations of granular media"
7. DFG-STW 12272 "Hydrodynamic theory of wet particle systems"
8. STW Take-Off Phase 1 "The MercuryLab Project"
9. STW VIDI 13472 "Shaping Segregation"
10. STW 15050 "Multiscale modelling of agglomeration - Application to tableting and selective laser sintering"
11. NWO VIDI 16604 "Virtual Prototyping of Particulate Processes"

We would like to thank MercuryLab for their financial support, contribution to the code and the development of the training material for MercuryDPM. MercuryLab is the official provider of consultancy, cloud interfaces and training for MercuryDPM and its associated tools. All MercuryLab training material is freely available under the CC-BY licence.

References

[1] A.R. Thornton, D. Krijgsman, A. te Voortwis, V. Ogarko, S. Luding, R. Fransen, S. Gonzalez, O. Bokhove, O. Imole, T. Weinhart, Discrete Elem. Methods 6 (2013).
[2] A.R. Thornton, D. Krijgsman, R.H.A. Fransen, S.G. Briones, D.R. Tunuguntla, A. te Voortwis, S. Luding, O. Bokhove, T. Weinhart, EnginSoft Simul.-Based Eng. Sci. 10 (1) (2013) 48–53.
[3] T. Weinhart, D.R. Tunuguntla, M. van Schrojenstein-Lantman, A. van der Horn, I.F.C. Denissen, C.R. Windows-Yule, A.C. de Jong, A.R. Thornton, Proc. 7th Int. Conf. Discrete Element Methods, Springer, 2016, pp. 1353–1360.
[4] T. Weinhart, D.R. Tunuguntla, M.P.V.S. Lantman, I.F. Denissen, C.R.W. Yule, H. Polman, J.M. Tsang, B. Jin, L. Orefice, K. van der Vaart, S. Roy, H. Shi, A. Pagano, W. den Breeijen, B. Scheper, A. Jarray, S. Luding, A.R. Thornton, Int. Conf. Particle-Based Methods V, 2017, pp. 123–134.
[5] P.A. Cundall, O.D.L. Strack, Géotechnique 29 (1) (1979) 47–65.
[6] T. Pöschel, T. Schwager, Computational Granular Dynamics: Models and Algorithms, Springer, 2005.
[7] D. Palanisamy, Micro-Hydrodynamics of Non-Spherical Colloids: a Brownian Dynamics Study (Ph.D. thesis), University of Twente, Enschede, The Netherlands, 2019.
[8] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, LAPACK Users' Guide, third ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, ISBN: 0-89871-447-8, 1999, (paperback).
[9] V. Ogarko, S. Luding, Comput. Phys. Comm. 183 (4) (2012) 931–936.
[10] D. Krijgsman, V. Ogarko, S. Luding, Comput. Part. Mech. 1 (3) (2014) 357–372.
[11] T. Weinhart, A.R. Thornton, S. Luding, O. Bokhove, Granul. Matter 14 (2) (2012) 289–294.
[12] T. Weinhart, R. Hartkamp, A.R. Thornton, S. Luding, Phys. Fluids 25 (7) (2013) 070605.
[13] D.R. Tunuguntla, A.R. Thornton, T. Weinhart, Comput. Part. Mech. 3 (3) (2016) 349–365.
[14] T. Weinhart, C. Labra, S. Luding, J.Y. Ooi, Powder Technol. 293 (2016) 138–148.
[15] M.P. Allen, D.J. Tildesley, Computer Simulation of Liquids, Oxford University Press, 1989.
[16] A.R. Thornton, T. Weinhart, V. Ogarko, S. Luding, Comput. Methods Mater. Sci. 13 (2) (2013) 197–212.
[17] S. Raschdorf, M. Kolonko, Internat. J. Numer. Methods Engrg. 85 (2011) 625–639.
[18] O.I. Imole, D. Krijgsman, T. Weinhart, V. Magnanimo, B.E.C. Montes, M. Ramaioli, S. Luding, Powder Technol. 293 (2016) 69–81.
[19] R. Kawamoto, E. Andò, G. Viggiani, J.E. Andrade, J. Mech. Phys. Solids 111 (2018) 375–392.
[20] L. Piegl, W. Tiller, The NURBS Book, Springer, 2012.
[21] J. Bel, D. Branque, H. Wong, G. Viggiani, N. Losacco, ITA-AITES World Tunnel Congress 2016, WTC 2016, Vol. 4, 2016, pp. 3219–3229.
[22] D.R. Tunuguntla, T. Weinhart, A.R. Thornton, Comput. Part. Mech. 4 (4) (2017) 387–405.
[23] S. Rubio-Largo, F. Alonso-Marroquin, T. Weinhart, S. Luding, R. Hidalgo, Physica A 443 (2016) 477–485.
[24] D. Tunuguntla, T. Weinhart, A. Thornton, Alert doctoral school 2017 discrete element modeling, 2017, p. 181.
[25] S. Roy, B.J. Scheper, H. Polman, A.R. Thornton, D.R. Tunuguntla, S. Luding, T. Weinhart, Eur. Phys. J. E 42 (2) (2019) 14.
[26] K. van der Vaart, M. van Schrojenstein Lantman, T. Weinhart, S. Luding, C. Ancey, A.R. Thornton, Phys. Rev. Fluids 3 (7) (2018) 074303.
[27] H. Shi, S. Roy, T. Weinhart, V. Magnanimo, S. Luding, Granul. Matter 22 (14) (2020).
[28] A. Lees, S. Edwards, J. Phys. C 5 (15) (1972) 1921.
[29] D. Pan, J. Hu, X. Shao, Mol. Simul. 42 (4) (2016) 328–336.
[30] H. Kobayashi, R. Yamamoto, J. Chem. Phys. 134 (6) (2011) 064110.
[31] I.F.C. Denissen, T. Weinhart, A. Te Voortwis, S. Luding, J.M.N.T. Gray, A.R. Thornton, J. Fluid Mech. 866 (2019) 263–297.
[32] S. Luding, Granul. Matter 10 (4) (2008) 235.
[33] J. Tomas, Powder Handl. Process. 15 (5) (2003) 296–314.
[34] R. Fuchs, T. Weinhart, M. Ye, S. Luding, H.-J. Butt, M. Kappl, EPJ Web of Conferences, Vol. 140, EDP Sciences, 2017, p. 13012.
[35] T. Weinhart, M. Post, I.F.C. Denissen, D.R. Tunuguntla, E. Grannonio, N. Losacco, J.M.F. Tsang, J. Barbosa, W. den Otter, A.R. Thornton, 8th Int. Conf. Discrete Element Methods, Springer, 2019.
[36] S. Roy, A. Singh, S. Luding, T. Weinhart, Comput. Part. Mech. 3 (4) (2016) 449–462.
[37] S. Roy, S. Luding, T. Weinhart, Phys. Rev. E 98 (5) (2018) 052906.
[38] A.G. Pagano, V. Magnanimo, T. Weinhart, A. Tarantino, Géotechnique, 0, (0) 1, https://doi.org/10.1680/jgeot.18.P.060.
[39] Tribology-ABC, Hertzian contacts, 2015, http://www.tribology-abc.com/sub10.htm, [Online; accessed 01 August 2015].
[40] C. Thornton, J. Appl. Mech. 64 (2) (1997) 383–386.
[41] S. Luding, Phys. Dry Granul. Media - NATO ASI Ser. E350 (ISSN: 0168-132X) 1 (1998) 285.
[42] R.L. Jackson, I. Green, D.B. Marghitu, Nonlinear Dynam. 60 (3) (2010) 217–229.
[43] C. Thornton, S.J. Cummins, P.W. Cleary, Powder Technol. 233 (2013) 30–46.
[44] E.R. Johnston, Vector Mechanics for Engineers: Statics and Dynamics, Vol. 1, Tata McGraw-Hill Education, 2009.
[45] O.R. Walton, R.L. Braun, J. Rheol. (1978-present) 30 (5) (1986) 949–980.
[46] O.R. Walton, Mech. Mater. 16 (1) (1993) 239–247.
[47] J. Blendell, Encyclopedia of Materials: Science and Technology, Elsevier, Oxford, ISBN: 978-0-08-043152-9, 2001, pp. 8745–8750.
[48] T. Weinhart, R. Fuchs, T. Staedler, M. Kappl, S. Luding, Particles in Contact: Micro Mechanics, Micro Process Dynamics and Particle Collective, Springer, 2019, pp. 311–338.

[49] R. Fuchs, T. Weinhart, J. Meyer, H. Zhuang, T. Staedler, X. Jiang, S. Luding, MRS Proc. 1652 (2014) mrsf13–1652–ll07–07.
[50] R. Fuchs, M. Ye, T. Weinhart, S. Luding, H.-J. Butt, M. Kappl, International Congress on Particle Technology (ParTec), 2016.
[51] M.Y. Shaheen, A.R. Thornton, S. Luding, T. Weinhart, Int. Conf. Discrete Element Methods (DEM8), 2019.
[52] Y. Gan, P. Rognon, I. Einav, Phil. Mag. 92 (28–30) (2012) 3405–3417.
[53] R. Fuchs, J. Meyer, T. Staedler, X. Jiang, Tribol. - Mater. Surf. Interfaces 7 (2) (2013) 103–107.
[54] S. Roy, S. Luding, T. Weinhart, New J. Phys. 19 (4) (2017) 043014.
[55] C.D. Willett, M.J. Adams, S.A. Johnson, J.P. Seville, Langmuir 16 (24) (2000) 9396–9405.
[56] S. Roy, S. Luding, W.K. den Otter, A.R. Thornton, T. Weinhart, J. Fluid Mech. (2019) submitted for publication.
[57] R. Hibbeler, Engineering Mechanics: Statics & Dynamics, 14th Edn. Hoboken, Pearson Prentice Hall Pearson Education, Inc, NJ, 2016.
[58] A. Podlozhnyuk, S. Pirker, C. Kloss, Comput. Part. Mech. 4 (1) (2017) 101–118.
[59] I.F.C. Denissen, On Segregation in Bidisperse Granular Flows (Ph.D. thesis), University of Twente, Enschede, The Netherlands, 2019.
[60] S. He, J. Gan, D. Pinson, Z. Zhou, Powder Technol. 341 (2019) 157–166.
[61] R. Furukawa, K. Kadota, T. Noguchi, A. Shimosaka, Y. Shirakawa, AAPS Pharm. Sci. Tech. 18 (6) (2017) 2368–2377.
[62] A. Skelbæk-Pedersen, T. Vilhelmsen, V. Wallaert, J. Rantanen, J. Pharm. Sci. 108 (3) (2019) 1246–1253.
[63] D. Mašín, Eng. Geol. 165 (2013) 73–88.
[64] M.P. van Schrojenstein Lantman, A Study on Fundamental Segregation Mechanisms in Dense Granular Flows (Ph.D. thesis), University of Twente, Enschede, The Netherlands, 2019.
[65] M. Heil, A.L. Hazel, Fluid-Structure Interaction, Springer, 2006, pp. 19–49.
[66] M. Dratt, A. Katterfeld, Granul. Matter 19 (3) (2017) 49.
[67] G. Ma, W. Zhou, X.-L. Chang, W. Yuan, Int. J. Geomech. 14 (4) (2013) 04014014.
[68] H. Cheng, H. Yamamoto, K. Thoeni, Y. Wu, Geotext. Geomembr. 45 (4) (2017) 361–376.
[69] H. Cheng, H. Yamamoto, N. Guo, H. Huang, Proc. 7th Int. Conf. Discrete Element Methods, Springer Singapore, Singapore, 2017, pp. 445–453.
[70] T. Anderson, R. Jackson, Ind. Eng. Chem. Fundam. 6 (4) (1967) 527–539.
[71] C. Davies, J. Aerosol Sci. 10 (5) (1979) 477–513.
[72] H. Cheng, T. Shuku, K. Thoeni, H. Yamamoto, Granul. Matter 20 (1) (2018) 11.
[73] H. Cheng, T. Shuku, K. Thoeni, P. Tempone, S. Luding, V. Magnanimo, Comput. Methods Appl. Mech. Engrg. 350 (2019) 268–294.
[74] A.R. Thornton, M. Post, L. Orefice, P. Rapino, S. Roy, H. Polman, M.Y. Shaheen, J. Alvarez Naranjo, H. Cheng, L. Jing, H. Shi, J. Mbaziira, R. Roeplal, T. Weinhart, 8th Int. Conf. Discrete Element Methods, Springer, 2019.