
ExaDEM & ExaNBody

Raphaël Prat, Thierry Carrard, Paul Lafourcade

Aug 02, 2026

EXADEM VARIANT



EXADEM SOFTWARE

1.1 Overview of ExaDEM

ExaDEM is a software solution in the field of computational simulations. It's a Discrete Element Method (DEM) code developed within the `exaNBody` framework. This framework provides the basis for DEM functionalities and performance optimizations. A notable aspect of ExaDEM is its hybrid parallelization approach, which combines the use of MPI (Message Passing Interface) and threads (OpenMP). This combination aims to enhance computation times for simulations, making them more efficient and manageable.

Additionally, ExaDEM offers compatibility with MPI + GPUs, using the CUDA programming model (Onika layer). This feature provides the option to leverage GPU processing power for potential performance gains in simulations. Written in C++17, ExaDEM is built on a contemporary codebase. It aims to provide researchers and engineers with a tool for addressing DEM simulations.

1.2 ExaDEM Capabilities

The following table provides a glossary of features supported by ExaDEM, categorized by their computational platform (CPU or GPU) and particle type (Sphere or Polyhedron). Please note that this information is indicative and may be subject to change based on updates or modifications to the ExaDEM framework.

Table 1: Glossary Of Features

Features	Sphere(CPU)	Sphere(GPU)	Polyhedron(CPU)	Polyhedron(GPU)
Field Set Plugin	✓	✓	✓	✓
I/O Files	✓		✓	
Paraview Files	✓		✓	
Particle Migration (MPI)	✓		✓	
Load Balancing (RCB)	✓		✓	
Numerical Scheme Plugin	✓	✓	✓	✓
Regions	✓	✓	✓	✓
Particles Generator	✓		✓	
Contact Law	✓	✓	✓	✓
Gravity Force	✓	✓	✓	✓
Quadratic Drag Force	✓	✓	✓	✓
Driver Cylinder	✓	✓	✓	✓
Driver Wall/Surface	✓	✓	✓	✓
Driver type Ball/Sphere	✓	✓	✓	✓
Driver type RShape	✓	✓	✓	✓
Contact Network File	✓		✓	
Fragmentation			✓	✓

Note

With version 1.0.1, `exaDEM` can now run GPU calculations with polyhedra. Nevertheless, optimizations are required to achieve good performance. These are currently being implemented.

1.3 Default Execution Graph With ExaDEM

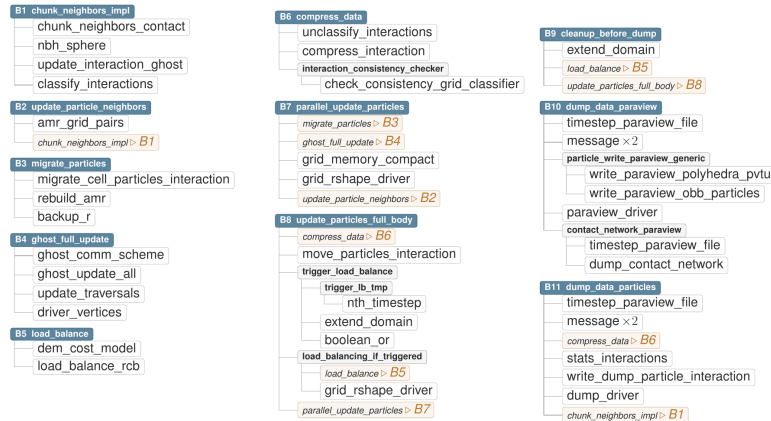
`ExaDEM` is based on the operator execution system implemented by the `ExaNBody` framework. In particular, this allows us to define a set of default operators for `ExaDEM` to run a DEM simulation following a default Verlet Vitesse integration scheme. `ExaDEM` offers two variants, depending on whether you want to simulate spherical particles only or spheropolyhedron particles. To choose between these two variants, you'll need to include either `config_spheres.msp` or `config_polyhedra.msp` at the beginning of your input file. If neither is defined, the simulation will not work. Including these files allows you to override certain operators left at *nop* (for no operation). The graph below shows the sequence of `exaDEM`'s main operators for version 1.2.2.

ExaDEM — simulation call graph — Rotating Drum Example.

Operator tree in execution order; indentation = sub-operators (callees).
Bold node = calls sub-operators. $\times Bn$ = expands to reusable building block n (shown bottom). $\times N$ = N consecutive identical calls.
 Phases: ● **sim** ● **first_iteration** ● **loop** ● **simulation_epilog** ● building block

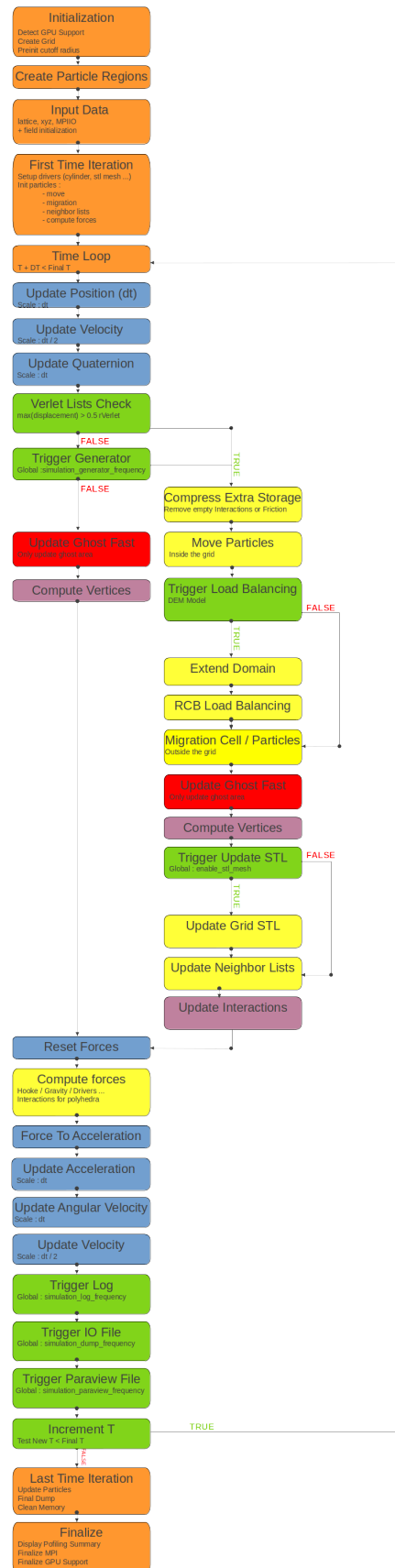


Building blocks (reusable operator subtrees, referenced above by $\times Bn$)



[View the previous version of the graph](#)

This image corresponds to an older version of the execution graph.



Note

- Violet: These operators are only called with the polyhedra variant.
- Blue: Numerical time scheme operators.

INSTALLING EXADEM

ExaDEM provides flexible installation methods to meet various user and developer needs. For user usage, the Spack (version 1.1.0) package manager offers a straightforward installation process, ideal for user-only. However, for those who intend to develop or customize ExaDEM, CMake is recommended, as it supports both *CPU* and *GPU* configurations (*GPU* support is not available with Spack).

Choose the method below that best suits your setup and follow the instructions for a smooth installation experience.

2.1 Installation With CMake

2.1.1 Minimal Requirements

The first step involves the installation of `yaml-cpp`, which can be achieved using either the `spack` package manager or `cmake`:

`spack`

```
spack install yaml-cpp@0.6.3
spack load  yaml-cpp@0.6.3
```

`apt-get` install

```
sudo apt install libyaml-cpp-dev
```

`cmake`

```
export CURRENT_HOME=$PWD
git clone --depth 1 --branch  yaml-cpp-0.6.3 https://github.com/jbeder/yaml-cpp.git
mkdir ${CURRENT_HOME}/build-yaml-cpp && cd ${CURRENT_HOME}/build-yaml-cpp
cmake ../yaml-cpp/ -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=${CURRENT_HOME}/
↳install-yaml-cpp -DYAML_BUILD_SHARED_LIBS=ON -DYAML_CPP_BUILD_TESTS=OFF
make install -j 4
cd ${CURRENT_HOME}
export PATH_TO_YAML=$PWD/install-yaml-cpp
```

Please ensure to remove `yaml-cpp` and `build-yaml-cpp`. When installing Onika, `exaNBody`, and `exaDEM`, remember to add the following to your `cmake` command: `-DCMAKE_PREFIX_PATH=${PATH_TO_YAML}`.

To proceed with the installation, your system must meet the minimum prerequisites (MPI, Onika, and `ExaNBody`). The next step involves the installation of Onika and `ExaNBody`:

Ubuntu CPU

```
export CURRENT_HOME=$PWD
git clone --branch v1.1.0 https://github.com/Collab4exaNBody/onika.git
git clone --branch v2.1.0 https://github.com/Collab4exaNBody/exaNBody.git
mkdir ${CURRENT_HOME}/build-onika && cd ${CURRENT_HOME}/build-onika
cmake ${CURRENT_HOME}/onika -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=${CURRENT_
→HOME}/install-onika -DONIKA_BUILD_CUDA=OFF
make install -j 10
export onika_DIR=${CURRENT_HOME}/install-onika
mkdir ${CURRENT_HOME}/build-exanb && cd ${CURRENT_HOME}/build-exanb
cmake ${CURRENT_HOME}/exaNBody -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=$
→{CURRENT_HOME}/install-exanb -DXNB_PARTICLE_TYPE_INT=uint32_t
make install -j 10
export exaNBody_DIR=${CURRENT_HOME}/install-exanb
rm -rf ${CURRENT_HOME}/build-onika
rm -rf ${CURRENT_HOME}/build-exanb
```

Ubuntu GPU

Please select the correct compute capability for your GPU for `DCMAKE_CUDA_ARCHITECTURES` instead of 86 in this example.

```
export CURRENT_HOME=$PWD
git clone --branch v1.1.0 https://github.com/Collab4exaNBody/onika.git
git clone --branch v2.1.0 https://github.com/Collab4exaNBody/exaNBody.git
mkdir ${CURRENT_HOME}/build-onika && cd ${CURRENT_HOME}/build-onika
cmake ${CURRENT_HOME}/onika -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=${CURRENT_
→HOME}/install-onika -DONIKA_BUILD_CUDA=ON -DCMAKE_CUDA_ARCHITECTURES=86
make install -j 10
export onika_DIR=${CURRENT_HOME}/install-onika
mkdir ${CURRENT_HOME}/build-exanb && cd ${CURRENT_HOME}/build-exanb
cmake ${CURRENT_HOME}/exaNBody -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=$
→{CURRENT_HOME}/install-exanb -DXNB_PARTICLE_TYPE_INT=uint32_t
make install -j 10
export exaNBody_DIR=${CURRENT_HOME}/install-exanb
rm -rf ${CURRENT_HOME}/build-onika
rm -rf ${CURRENT_HOME}/build-exanb
```

TOPAZE GPU

Please note that you need to copy the onika and exaNBody repositories on topaze. You can load the YAML module such as `module load yaml-cpp/` and add this option to the cmake command: `-DCMAKE_PREFIX_PATH=/ccc/products/yaml-cpp-0.6.3/system/default/`

```
module load gnu/13.2.0 cuda/12.4 mpi/openmpi/4.1.6 cmake/3.29.6
cd $CCSCRATCHDIR
export CURRENT_HOME=$PWD
# copy onika v1.1.0
# copy exaNBody v2.1.0
mkdir ${CURRENT_HOME}/build-onika && cd ${CURRENT_HOME}/build-onika
cmake ${CURRENT_HOME}/onika -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=${CURRENT_
→HOME}/install-onika -DONIKA_BUILD_CUDA=ON -DCMAKE_CUDA_ARCHITECTURES=80
make install -j 10
export onika_DIR=${CURRENT_HOME}/install-onika
mkdir ${CURRENT_HOME}/build-exanb && cd ${CURRENT_HOME}/build-exanb
```

(continues on next page)

(continued from previous page)

```
cmake ${CURRENT_HOME}/exaNBody -DCMAKE_BUILD_TYPE=Release -DCMAKE_INSTALL_PREFIX=$
↪ ${CURRENT_HOME}/install-exanb -DXNB_PARTICLE_TYPE_INT=uint32_t
make install -j 10
export exaNBody_DIR=${CURRENT_HOME}/install-exanb
rm -rf ${CURRENT_HOME}/build-onika
rm -rf ${CURRENT_HOME}/build-exanb
```

2.1.2 Optional Dependencies

Before proceeding further, you have the option to consider the following dependencies:

- CUDA
- HIP
- RSA_MPI

2.1.3 ExaDEM Installation

To install ExaDEM, follow these steps:

Set the exaNBody_DIR environment variable to the installation path. Clone the ExaDEM repository using the command:

```
git clone https://github.com/Collab4exaNBody/exaDEM.git
```

Create a directory named build-exaDEM and navigate into it:

```
mkdir build-exaDEM && cd build-exaDEM
```

Run CMake to configure the ExaDEM build:

cmake Minimal

```
cmake ../exaDEM -DCMAKE_BUILD_TYPE=Release
make -j 4
source bin/setup-env.sh
```

cmake GPU (a100)

exaDEM itself does not take any GPU-specific cmake flag: GPU support is entirely inherited from onika/exaNBody. Make sure onika_DIR and exaNBody_DIR point to installations built with -DONIKA_BUILD_CUDA=ON and the correct -DCMAKE_CUDA_ARCHITECTURES (see the “Ubuntu GPU” tab above), then the command below is identical to the CPU build.

```
cmake ../exaDEM -DCMAKE_BUILD_TYPE=Release
make -j 4
source bin/setup-env.sh
```

Spack

See the “Installation With Spack” section below if you need to install spack itself.

```
git clone https://github.com/Collab4exaNBody/spack-repos.git
spack repo add spack-repos
spack install exadem
```

Note

Historical note (versions $\leq 1.1.1$ only): in these old releases, the maximum number of vertices per particle shape was set to 8 by default and could be changed by adding `-DEXADEM_MAX_VERTICES=N` to the `cmake` command. This option no longer exists in current versions of ExaDEM.

Warning

Please do not forget to load the `exaDEM` environment before running a job: `source bin/setup-env.sh`.

This command will display all plugins and related operators. Example:

```
+ exadem_force_fieldPlugin
  operator    cylinder_wall
  operator    gravity_force
  operator    contact_force
  operator    rigid_surface
+ exadem_ioPlugin
  operator    print_simulation_state
  operator    read_xyz
  operator    read_dump_particles
```

Here are a few examples on CEA supercomputers:

CCRT Topaze Milan

You need to specify “`export exaNBody_DIR=${path_to_exaNBody}`” and “`export onika_DIR=${path_to_onika}`”

```
module load yaml-cpp/0.6.3 gnu/13.2.0 mpi/openmpi/4.1.6
cmake ${path_to_exaDEM} -DCMAKE_BUILD_TYPE=Release
```

CCRT Topaze a100

You need to specify “`export exaNBody_DIR=${path_to_exaNBody}`” and “`export onika_DIR=${path_to_onika}`”

```
module load gnu/13.2.0 cuda/12.4 mpi/openmpi/4.1.6 cmake/3.29.6
cmake ${path_to_exaDEM} -DCMAKE_BUILD_TYPE=Release
```

CCRT Irene Skylake and Rome

You need to specify “`export exaNBody_DIR=${path_to_exaNBody}`” and “`export onika_DIR=${path_to_onika}`”

```
module load yaml-cpp/0.6.3 gnu/13.2.0 mpi/openmpi/4.1.6
cmake ${path_to_exaDEM} -DCMAKE_BUILD_TYPE=Release
```

2.1.4 Launch examples / ctest

A set of minimal test cases can be run using the following command (non-regression test) in your *build* repository:

```
ctest --test-dir example
```

You can also add `exaDEM` to your `bashrc` by adding an alias (please, replace `YOURDIR` by your build directory):

```
vi ~/.bashrc
alias exaDEM='~/YOURDIR/build/exaDEM'
```

Or just on your local environment:

```
alias exaDEM=$PWD/exaDEM
```

2.2 Installation With Spack

Installation with spack is preferable for people who don't want to develop in exaDEM. Only stable versions are added when you install ExaDEM with Spack.

Note

The main of ExaDEM will never be directly accessible via this installation method.

2.2.1 Installing Spack

```
git clone --depth=2 --branch=v1.1.0 https://github.com/spack/spack.git
export SPACK_ROOT=$PWD/spack
source ${SPACK_ROOT}/share/spack/setup-env.sh
```

2.2.2 Installing ExaDEM

First get the spack repository in the exaDEM directory and add it to spack. It contains two packages: exanbody and exadem:

```
git clone https://github.com/Collab4exaNBody/spack-repos.git
spack repo add spack-repos
```

Second install ExaDEM (this command will install cmake, yaml-cpp, onika and exanbody).

```
spack install exadem
```

2.3 Running your simulation

2.3.1 CMake

Now that you have installed the ExaDEM and exaNBody packages, you can create your simulation file in YAML format (refer to the 'example' folder or the documentation for each operator). Once this file is constructed, you can initiate your simulation using the following instructions.

```
export N_OMP=1
export N_MPI=1
export OMP_NUM_THREADS=$N_OMP
mpirun -n $N_MPI ./exaDEM test-case.msp
```

2.3.2 Spack

The ExaDEM executable has been created in the spack directory. You can run your simulation with your input file (*your_input_file.msp*) such as:

```
spack load exadem
exaDEM your_input_file.msp
```

2.4 Adding Optional Packages:

2.4.1 RSA

This external library is available here: https://github.com/MarcJos/RSA_MPI

Installation of *RSA_MPI* via *cmake*:

```
export MY_ROOT_DIR=$PWD
# module load gnu/13.2.0 cuda/12.4 mpi/openmpi/4.1.6 cmake/3.29.6 # Topaze
git clone https://github.com/MarcJos/RSA_MPI.git
mkdir ${MY_ROOT_DIR}/build-rsmpi && cd ${MY_ROOT_DIR}/build-rsmpi
cmake ${MY_ROOT_DIR}/RSA_MPI/ -DINTERFACE_PYTHON=OFF -DCMAKE_INSTALL_PREFIX=${MY_ROOT_
↳DIR}/install-rsmpi
make install -j 10
export rsa_mpi_DIR=${MY_ROOT_DIR}/install-rsmpi
cd ${MY_ROOT_DIR}
rm -r ${MY_ROOT_DIR}/RSA_MPI
rm -r ${MY_ROOT_DIR}/${MY_ROOT_DIR}/build-rsmpi
```

You can install it with the following commands:

```
export rsa_mpi_DIR=$DIR
cmake ../exaDEM -DUSE_RSA=ON
```

Installation of *RSA_MPI* via *spack*:

```
git clone --depth=2 --branch=v1.1.0 https://github.com/spack/spack.git
source spack/share/spack/setup-env.sh
git clone https://github.com/Collab4exaNBody/spack-repos.git
spack repo add spack-repos
spack external find openmpi
spack install rsmpi
```

You can use this option with the following commands:

```
export rsa_mpi_DIR=`spack location -i rsmpi`
cmake ../exaDEM -DUSE_RSA=ON
```

2.4.2 Rockable

This external library is available here: <https://github.com/richefeu/rockable>

Installation of *Rockable* via *spack*:


```
git clone --depth=2 --branch=v1.1.0 https://github.com/spack/spack.git
source spack/share/spack/setup-env.sh
git clone https://github.com/Collab4exaNBody/spack-repos.git
spack repo add spack-repos
spack external find opengl
spack install rockable+see
```

Use *Rockable*:

```
spack load rockable
rockable file.conf -j 10
see conf0
```


TEST CASES

You can explore various basic test cases located in the *example* directory. These test cases serve as illustrative examples of ExaDEM’s functionality and can assist you in understanding its behavior and capabilities.

If you would like to view your test cases with paraview, here are some instructions:

- For spheres, please use one of these two options:
 - “Point Gaussian” and adjust the radius (one type)
 - Select “Use Scale Array” and select radius as “Gaussian Scale Array” while unselecting “Use Scale Transfer Function” and put the field “Gaussian Radius” to 1.
- For polyhedra:
 - [Paraview For Polyhedra](#)
 - [Paraview With OBBs](#)
 - [Contact Network](#)
- Drivers:
 - Only stl meshes have paraview outputs
 - For cylinder/surface/sphere, use “Sources” -> “Geometric Shapes” -> “Cylinder” / “Plane” / “Sphere”

Note

Some examples can take a long time to run, that’s why most of these examples have a “minimal version”, mainly used for testing (CI).

3.1 Example Using Spheres

All examples with spheres can be found in the directory: `example/spheres/`.

3.1.1 Example 1: Rotating drum

A DEM simulation of a set of spheres in a periodic rotating drum. Through this simulation, we can observe how particles interact, collide, and move in response to the drum’s motion. This provides insights into phenomena like particle segregation, convection currents, and mixing patterns, contributing to improved understanding of granular material behavior in rotational scenarios. A second scenario without periodic conditions is available, with two rigid surfaces fixed to the end of the drum.

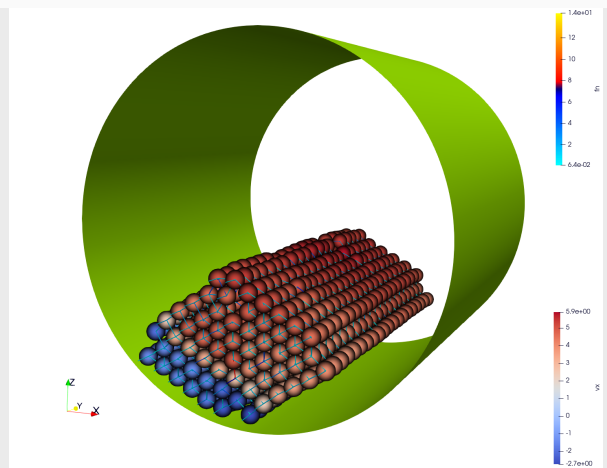
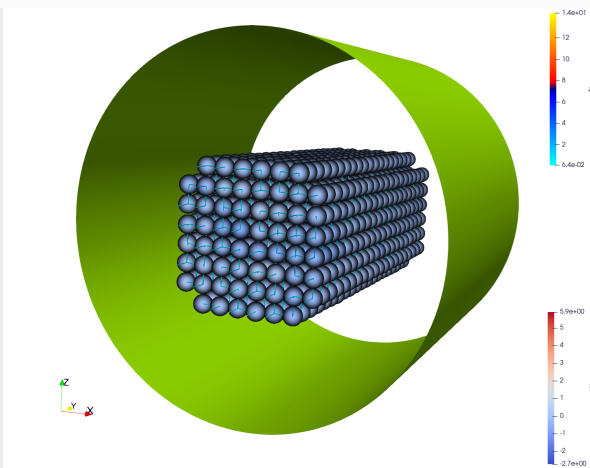
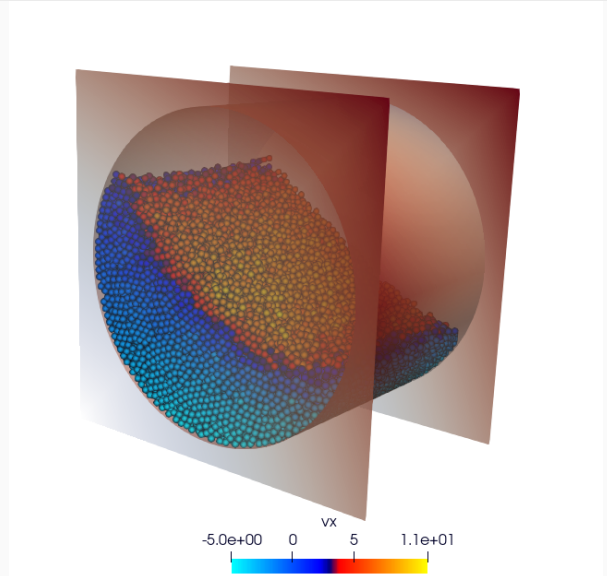
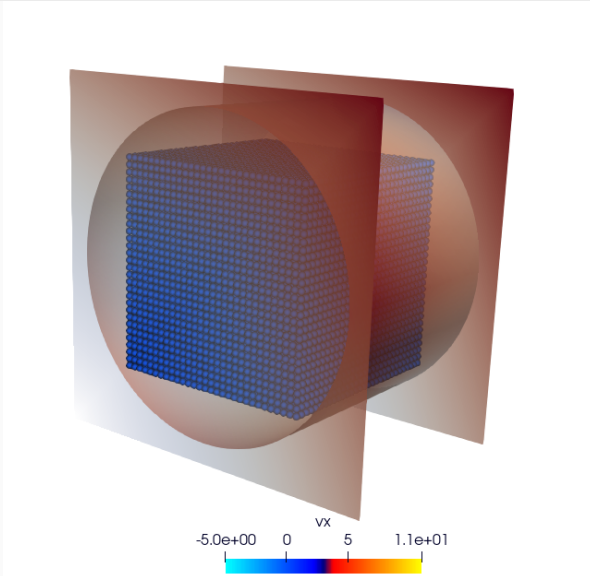
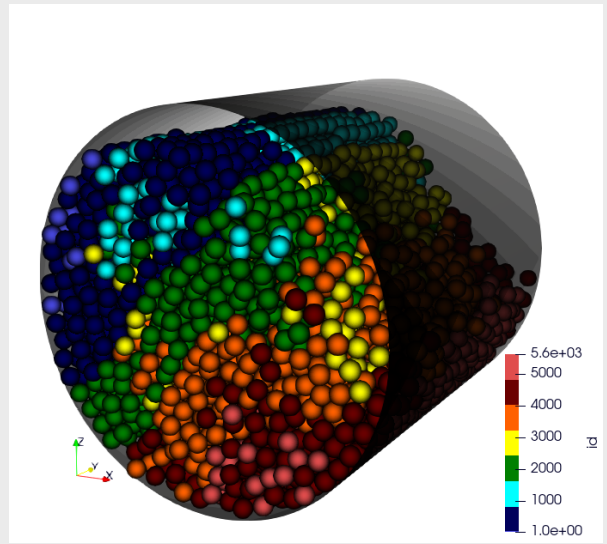
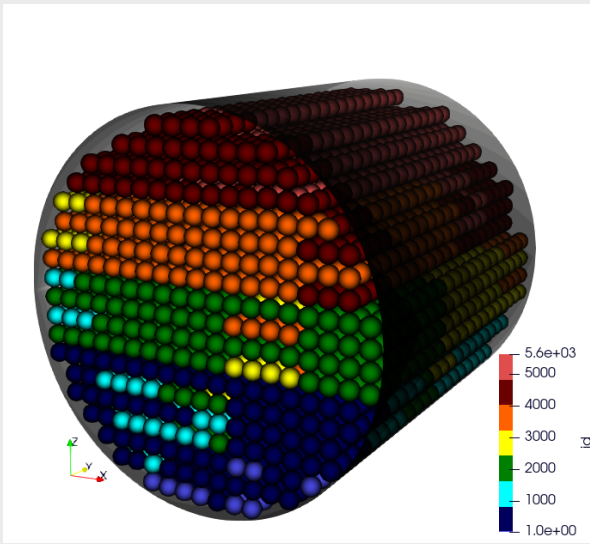
- Input files:
 - First test case: `example/spheres/rotating-drum/rotating-drum-periodic.msp`

- * 5,520 spheres
- * 20,000 timesteps
- * 15.4 seconds on 8 cores (OpenMP)
- Second test case: `example/spheres/rotating-drum/rotating-no-periodic.msp`
 - * 27,000 particles
 - * 20,000 timesteps
 - * 1 minute and 13 seconds on 8 cores (OpenMP)
- Third test case: `example/spheres/rotating-drum/rotating-periodic-cohesion.msp`
 - * 1,200 particles
 - * 30,000 timesteps
 - * 8 seconds on 12 cores (OpenMP)

Rotating Drum

Start

End

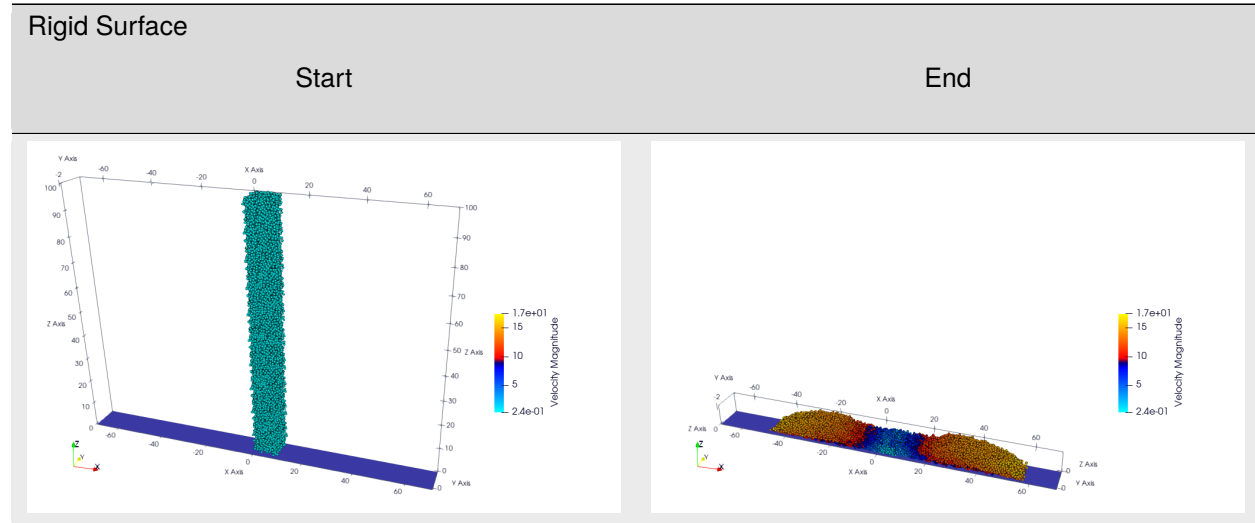


3.1. Example Using Spheres

3.1.2 Example 2: Rigid surface

A DEM simulation involving spherical particles falling onto a rigid surface offers a virtual exploration of particle dynamics in a gravity-driven scenario. This simulation captures the behavior of individual spherical particles as they fall and interact with a solid surface below.

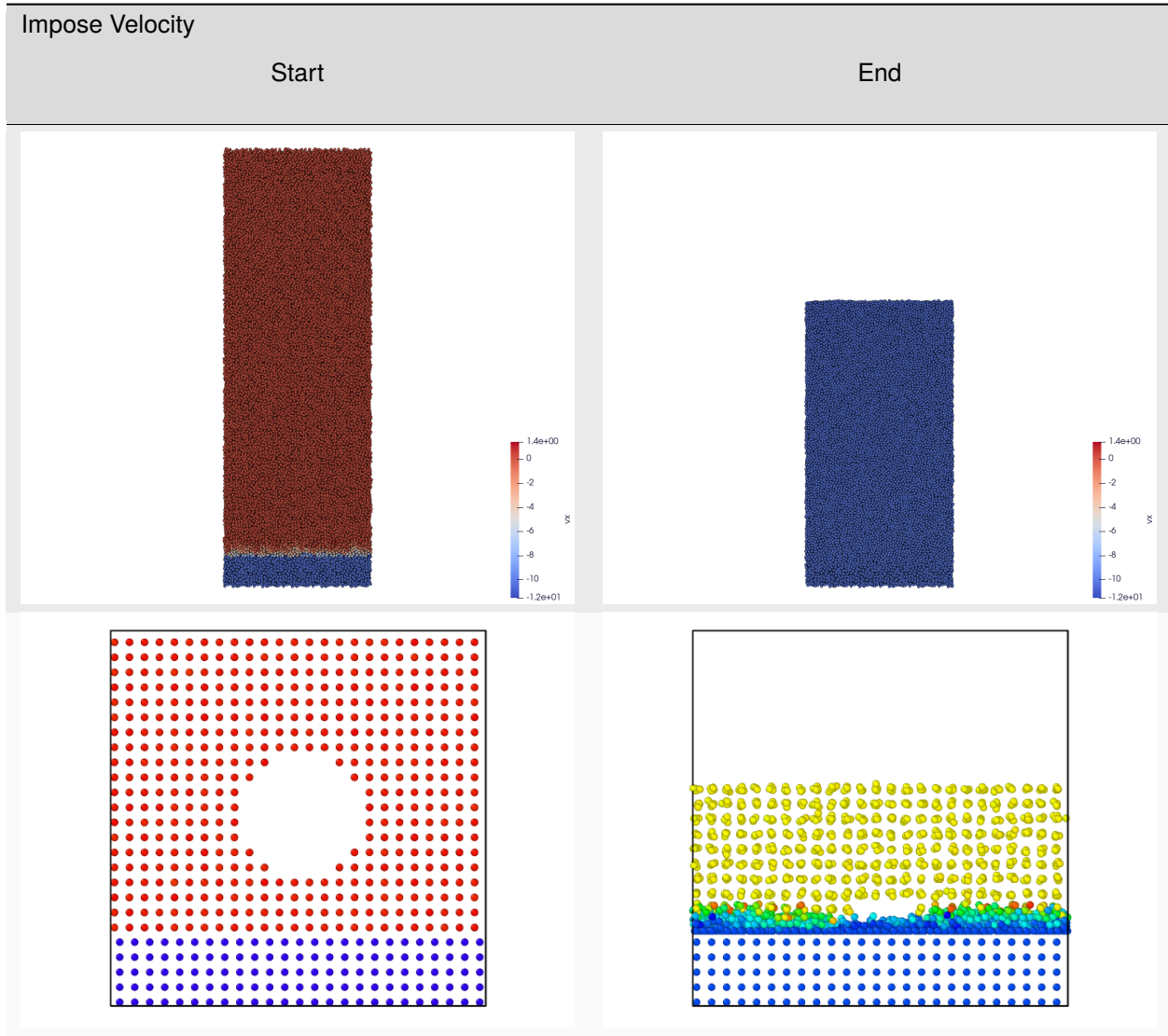
- Input file: `example/spheres/rigid-surface/rigid_surface.msp`
 - 7,360 particles
 - 10,000 timesteps
 - 6.6 seconds on 8 cores



3.1.3 Example 3: Impose Velocity

In this DEM simulation, a scenario is simulated where a group of particles with imposed velocity occupies a defined area. As other particles fall into this region, they interact with the moving particles, impacting their trajectories. The simulation provides insights into how moving driver particles influence the behavior of surrounding particles. Example: `impose_velocity_hole`, credit: `lafourcadp`.

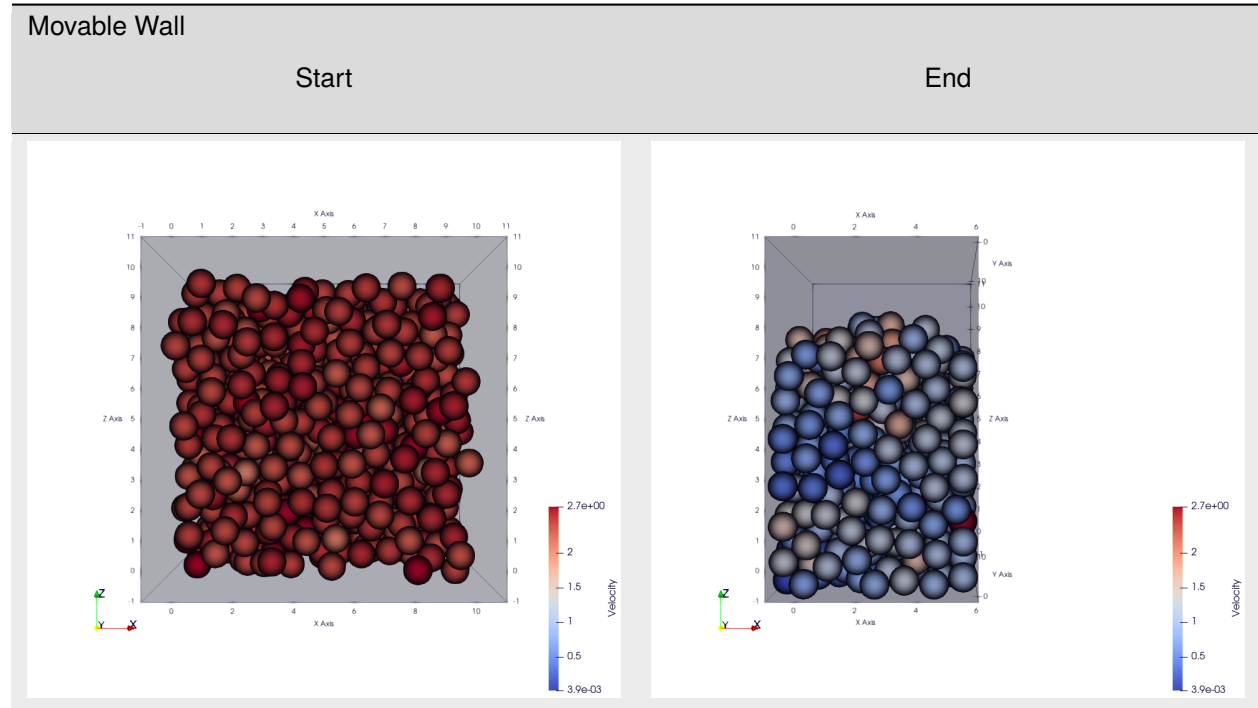
- Input files:
 - The first test case: `example/spheres/impose_velocity/impose_velocity_full.msp`
 - * 55,200 particles
 - * 10,000 timesteps
 - * 1 minute and 40 seconds on 8 cores (OpenMP)
 - The second test case: `example/spheres/impose_velocity/impose_velocity_hole.msp`
 - * 2,865 particles
 - * 2,000 timesteps
 - * 1.3 seconds on 8 cores (OpenMP)



3.1.4 Example 4: Movable wall

In this DEM simulation, a cluster of spherical particles is compacted against a rigid surface. A piston is introduced to apply a steadily increasing stress that linearly evolves over time. This simulation captures the dynamics as the piston's force gradually grows. As the piston imparts its stress, the particle block undergoes deformation and stress propagation.

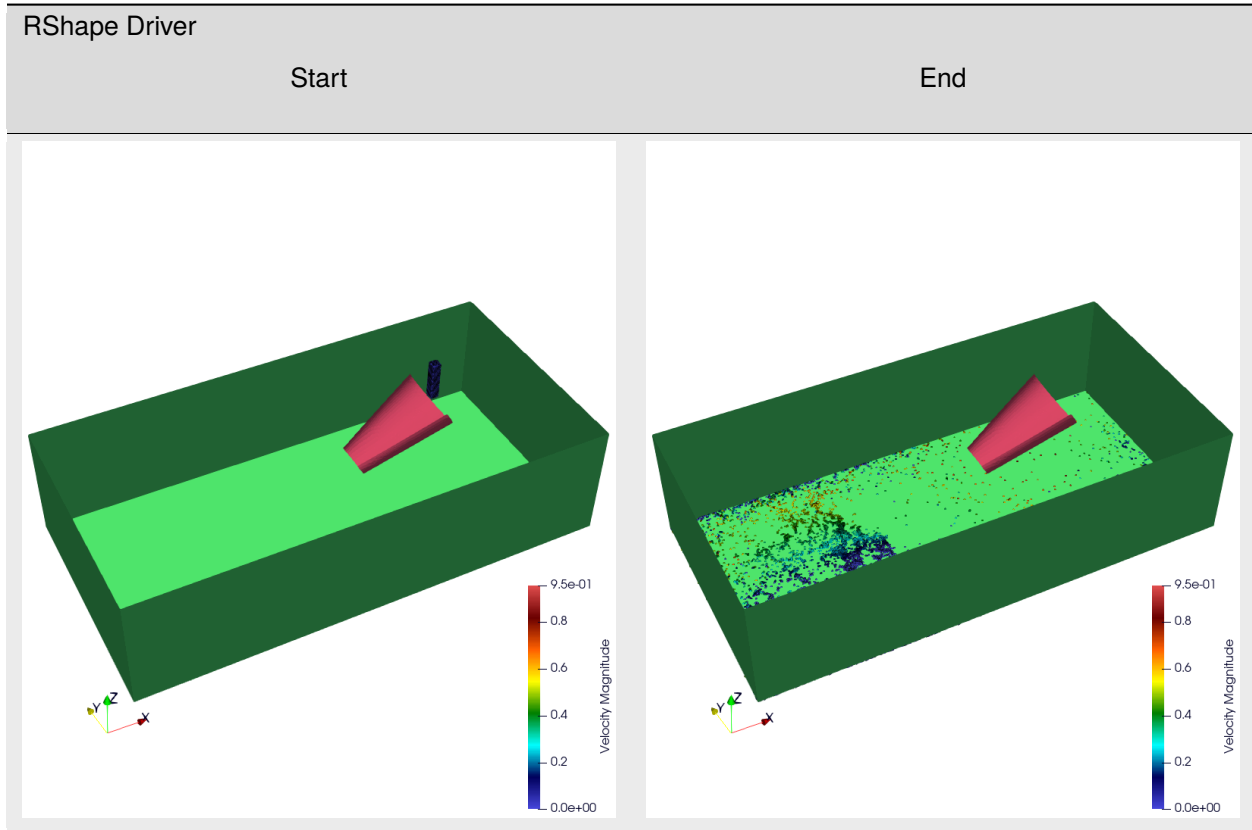
- Input file: `example/spheres/movable-wall/movable_wall.msp`
 - 736 particles
 - 30,000 timesteps
 - 4.1 seconds



3.1.5 Example 5: Using a RShape Driver (STL)

In this DEM simulation, a cluster of spherical particles falls onto an .stl mesh and into a box. This case study highlights the use of meshes containing numerous facets and allows adding all geometries coming from CAD or graphic design.

- Input file: `example/spheres/mesh-stl/mesh_stl.msp`
 - 5,000 particles
 - 20,000 timesteps
 - 36.4 seconds on 8 cores (OpenMP)



3.1.6 Example 6: Particle Generation With RSA Algorithm

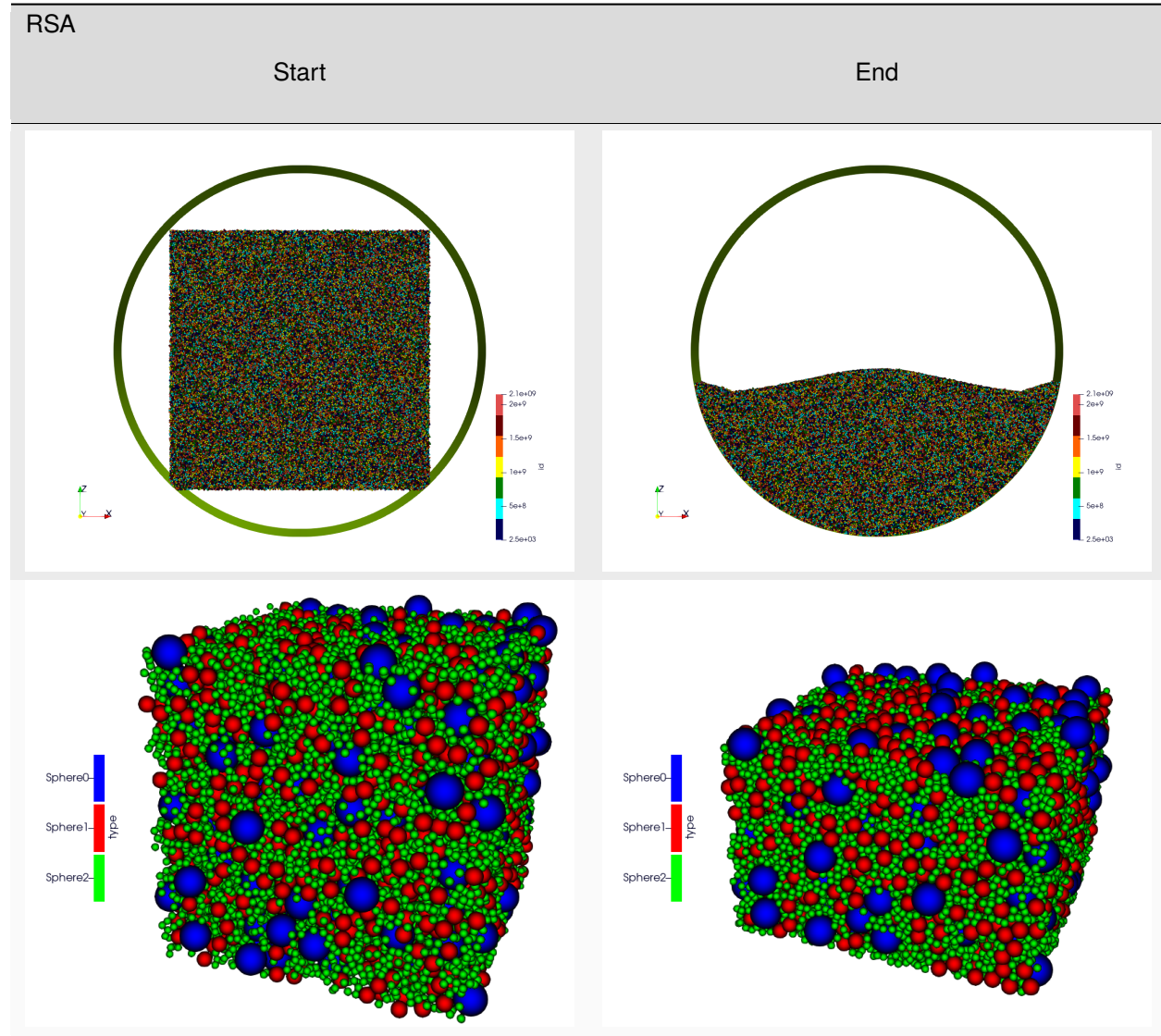
In this DEM simulation, a cluster of 287,642 spherical particles has been generated by the parallelized *rsa* algorithm [7]. Then, particles fall by gravity in a drum.

Input file: `example/spheres/rsa/rsa-full.msp`

A second example, available in `example/spheres/rsa/rsa_vol_frac.msp`, demonstrates how to define different volume fractions for particles of varying sizes.

Note

rsa is an optional package, please see installation guidelines to add it.



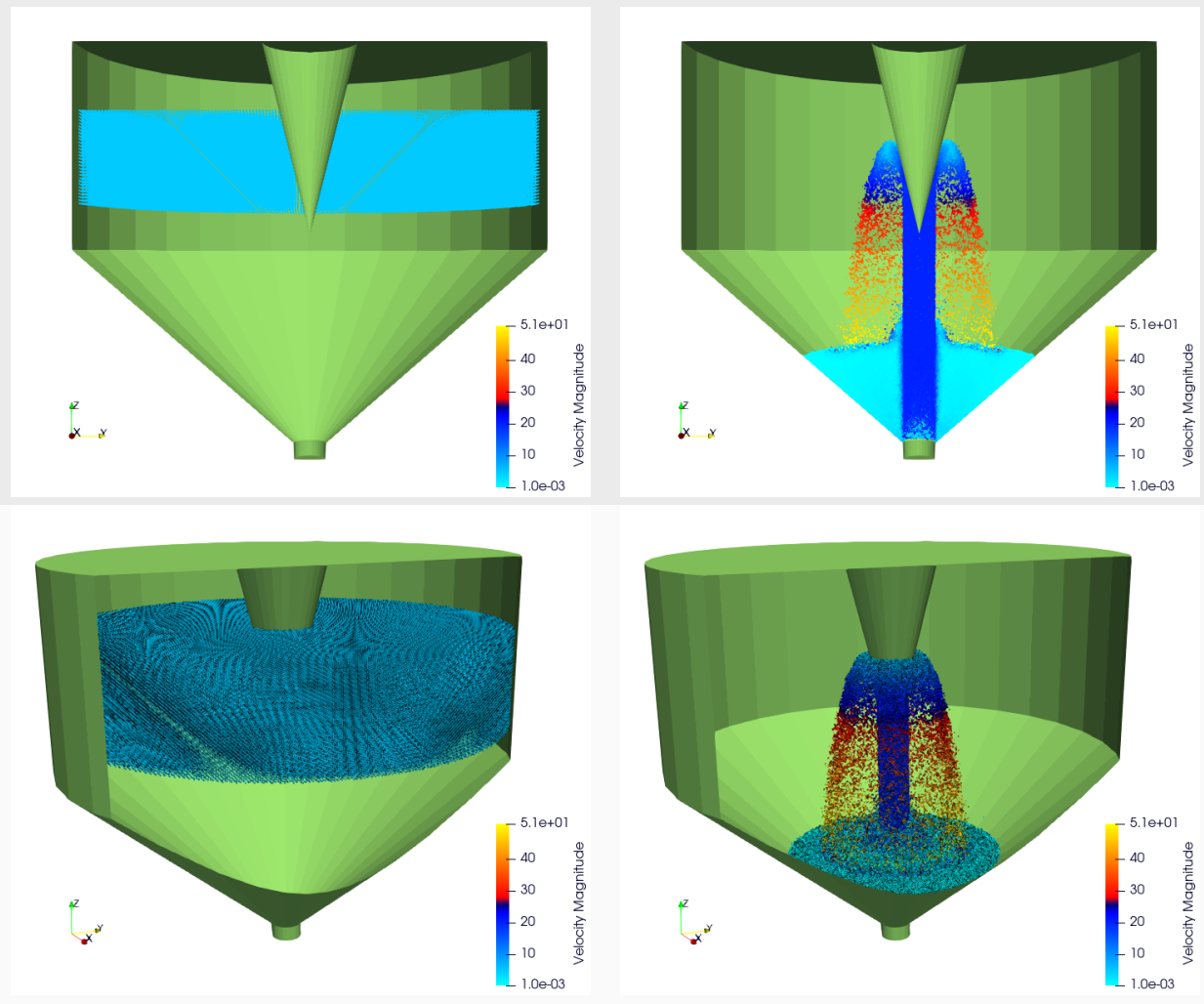
3.1.7 Example 7: Jet

This example demonstrates the application of a velocity field to spheres based on a Cartesian grid projection. Although it does not represent a physical scenario, a geyser-like effect has been simulated using a cylindrical shape, directing the particle velocities towards a specified speed. Future developments will involve applying non-uniform velocity fields to simulate more complex fluid configurations. Input files are available here: `example/spheres/jet/step1.msp` and `step2.msp` but they don't correspond exactly to the pictures (fewer particles).

Geyser Simulation

Start

End



3.1.8 Example 8: Mirror Boundary Conditions

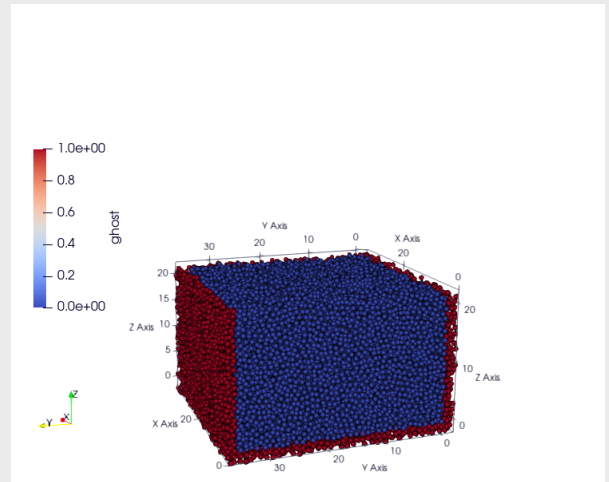
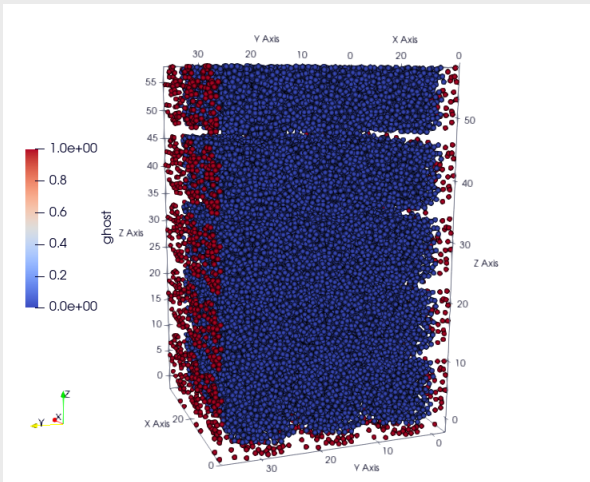
This example tests the mirror conditions available in exaNBody. Although these conditions are not directly applicable ("not physics"), because all fields are copied identically (without processing/filtering) at each time step in the ghost cells (e.g. velocity, moments), except for positions (axial symmetry). This example highlights this functionality and could potentially be coupled with other operators to develop new boundary conditions (e.g. resetting velocities to 0 to model a rigid surface). This example involves dropping 33,120 spheres, adding mirror boundary conditions in all directions, and letting them fall by gravity.

- Input file: `example/spheres/mirror/mirror.msp`.
 - 33,120 particles
 - 5,000 timesteps
 - 21.8 seconds on 8 cores (OpenMP)

Mirror Simulation

Start

End



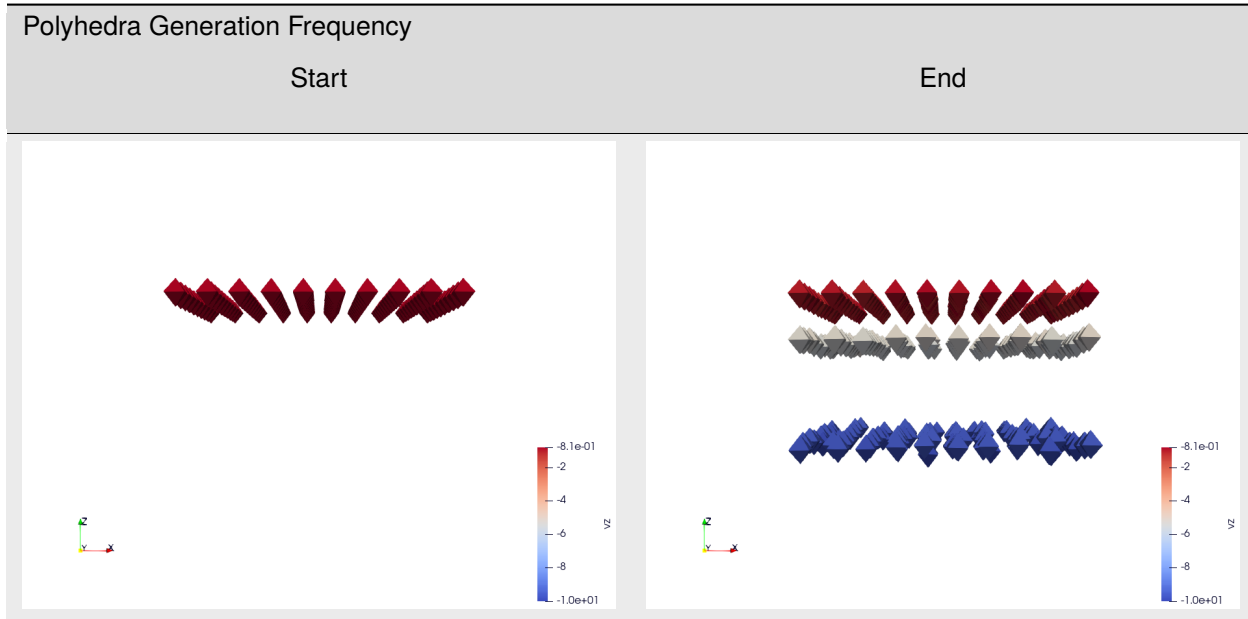
3.2 Examples Using Polyhedra

All examples with polyhedra can be found in the directory: `example/polyhedra/`.

3.2.1 Example 1: Polyhedra Generation Frequency

In this example, we simulate the generation of 100 new polyhedra at every 45000 time steps, representing their descent into a void environment. The primary objective is to illustrate the process of generating a lattice of polyhedra within a confined area. Additionally, we demonstrate the application of a series of operators to initialize various fields associated with the newly generated polyhedra. This example serves as a practical guide for setting up and executing simulations involving dynamic polyhedra generation and manipulation within defined spatial boundaries.

- Input file: `example/polyhedra/generator/generator.msp`
 - 100 then 200 and 300 Octahedra
 - 10,000 timesteps
 - 9.7 seconds on 8 cores (OpenMP)

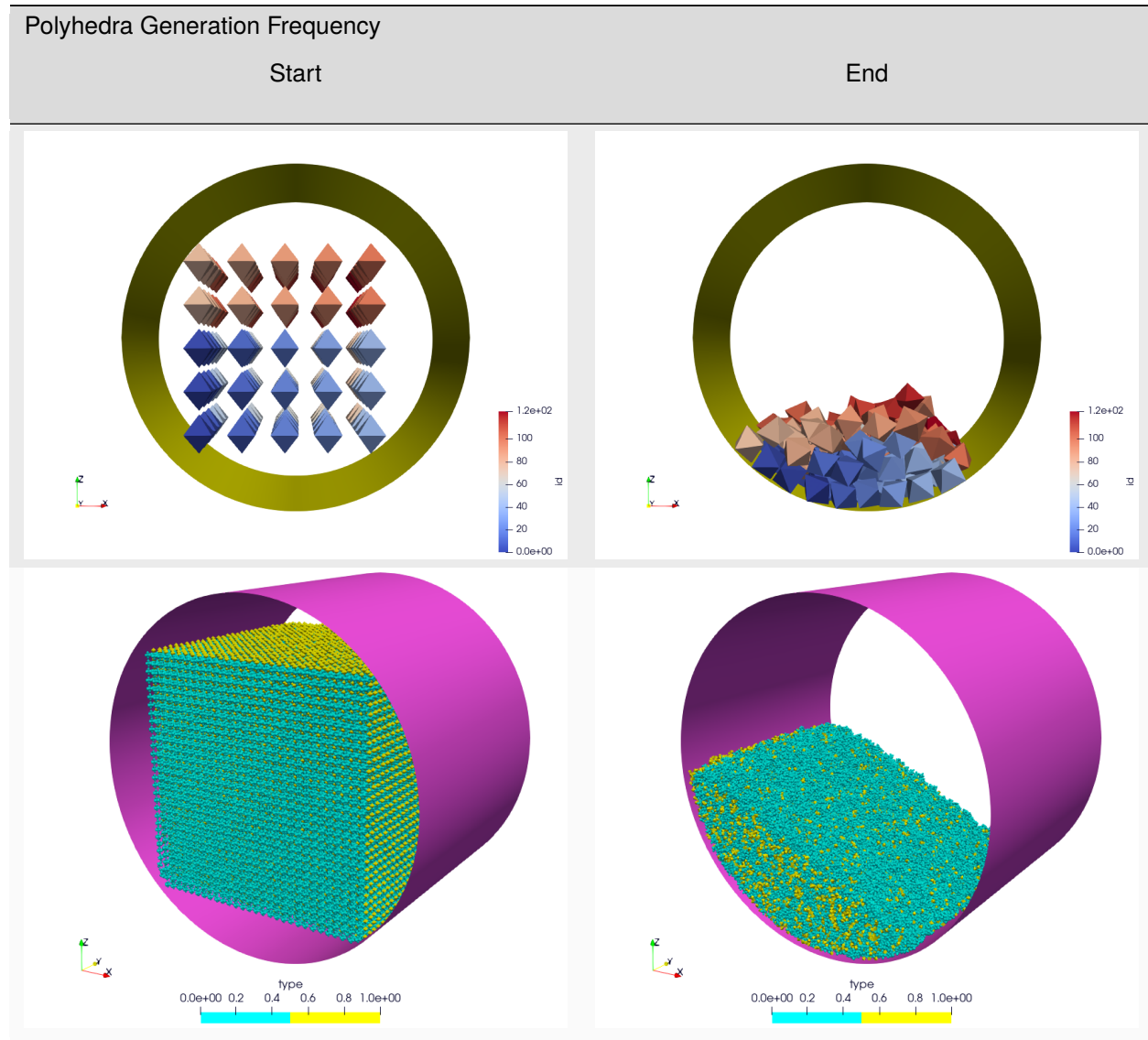


3.2.2 Example 2: Octahedra in a Rotating Drum

In this DEM simulation, we observe the dynamics of 125 octahedra as they descend into a rotating drum. The second test case contains 27,000 octahedra (yellow) and 27,000 hexapods (blue).

- Input files:
 - The first example: `example/polyhedra/rotating_drum/rotating-drum-mini.msp`
 - * 125 Octahedra
 - * 10,000 timesteps
 - * 1 second on 8 cores (OpenMP)
 - The second example: `example/polyhedra/rotating_drum/rotating-drum-mixte.msp`
 - * 27,000 Octahedra and 27,000 Hexapods
 - * 100,000 timesteps
 - * 41 minutes and 36 seconds on 8 cores (OpenMP)
 - * 6 minutes and 31 seconds on a100 (100,000 timesteps) with $rcut_inc = 0.4$
 - * 11 minutes and 24 seconds on a100 (200,000 timesteps) with $rcut_inc = 0.4$ (The stage where the particles fall is costly, as the particles are quickly moved from one cell to another, and the reconstruction of neighbor lists (on the CPU) is also costly.)
 - * The pictures correspond to the simulation after 1,000,000 timesteps
 - Other example: `example/polyhedra/rotating_drum/rotating-drum.msp`
 - * 5520 Octahedra
 - * 150,000 timesteps
 - * 2 minutes and 19 seconds on 8 cores (OpenMP)
 - Other example: `example/polyhedra/rotating_drum/rotating-drum-mixte-mini.msp`
 - * 125 Octahedra and 125 Hexapods

- * 20,000 timesteps
- * 2.5 seconds on 8 cores (OpenMP)

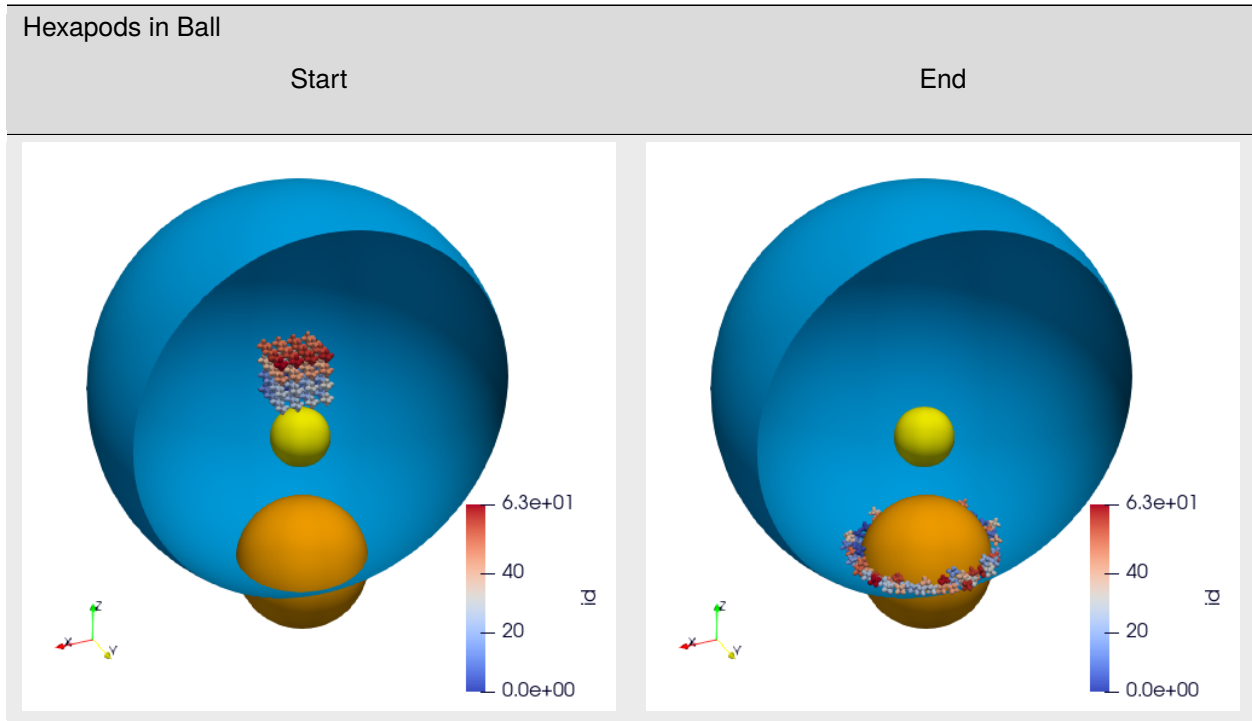


3.2.3 Example 3: Hexapods in a Ball

This DEM simulation example illustrates the gravitational descent of 64 hexapods within a large sphere. The primary environment consists of a spherical enclosure with a radius of 20 units and is centered at (2, 2, 0). As the hexapods descend under gravity within this enclosure, they encounter two additional spherical obstacles. The first obstacle (ball, represented as a small yellow ball with a radius of 3 units and centered at (2,2,-5). The second ball, centered at (2,2,-20) with a radius of 7 units, depicted as a large orange ball, intersects the surface of the primary blue sphere, adding complexity to the obstacle configuration. Through this simulation, exaDEM shows its capability to manage particle interactions with various obstacles (balls). Additionally, it showcases the versatility of drivers within the code, which can be employed to define both simulation boundary conditions and obstacles.

- Input file: `example/polyhedra/balls/balls.msp`
 - 64 Hexapods

- 150,000 timesteps
- 14.3 seconds on 8 cores (OpenMP)
- Other example: `example/polyhedra/balls/balls.msp`
 - 64 Hexapods
 - 25,000 timesteps
 - 2.7 seconds on 8 cores (OpenMP)



3.2.4 Example 4: Polyhedra With a RShape Driver (Box)

This simulation example illustrates the use of STL files with polyhedra. In this simulation, we drop a set of polyhedra (hexapods, octahedra, or both) by gravity into an open box to fill it completely.

- Input files:
 - First example: `example/polyhedra/stl_mesh/stl_mesh_box_hexapod.msp`
 - * 7,344 Hexapods
 - * 50,000 timesteps
 - * 2 minutes and 19 seconds on 8 cores (OpenMP)
 - Second example: `example/polyhedra/stl_mesh/stl_mesh_box_octahedron.msp`
 - * 2,430 Octahedra
 - * 50,000 timesteps
 - * 1 minute and 6 seconds on 8 cores (OpenMP)
 - Third example: `example/polyhedra/stl_mesh/stl_mesh_box_mixte.msp`
 - * 5760 Octahedra and 5760 Hexapods

- * 100,000 timesteps
- * 8 minutes and 36 seconds on 8 cores (OpenMP)
- * 2 minutes and 34 seconds on a GPU a100 ($rcut_inc = 0.2$, $cell_size = 6$, and $grid_dims = [3,3,10]$)

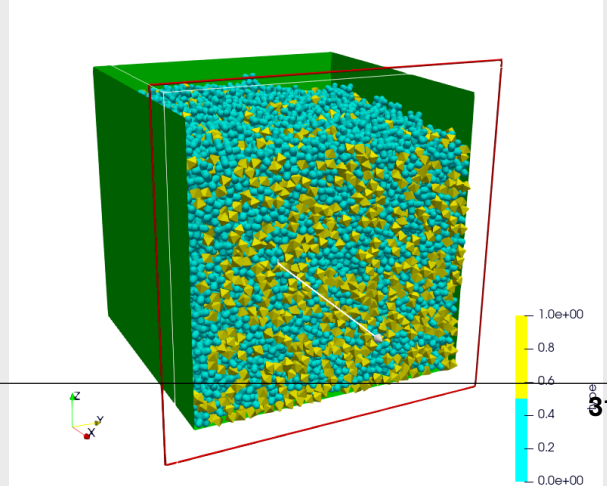
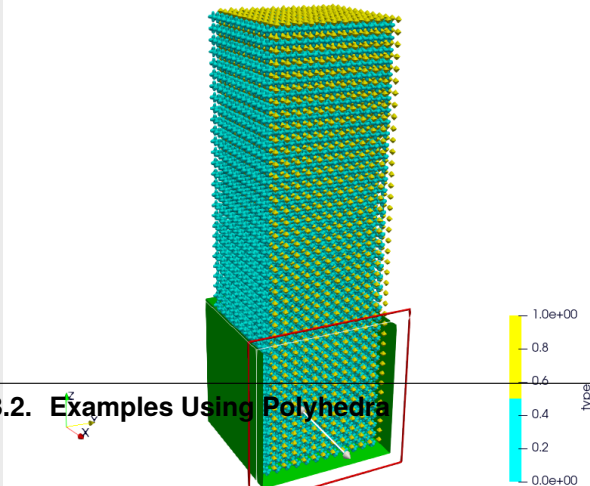
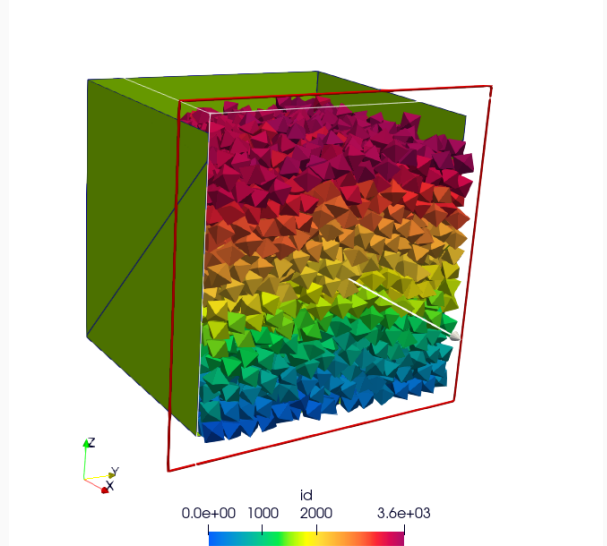
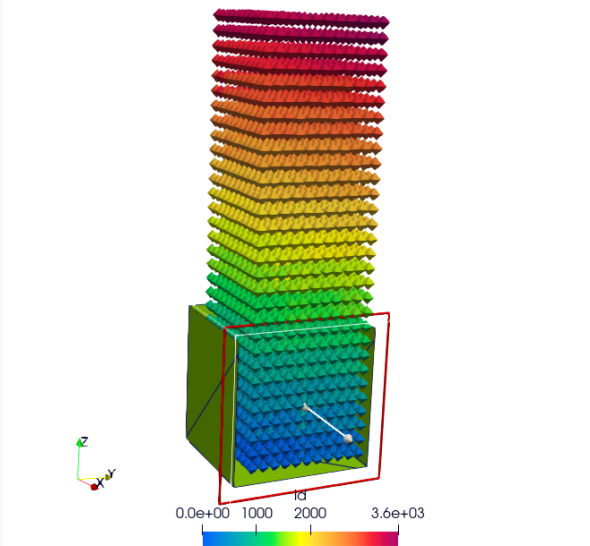
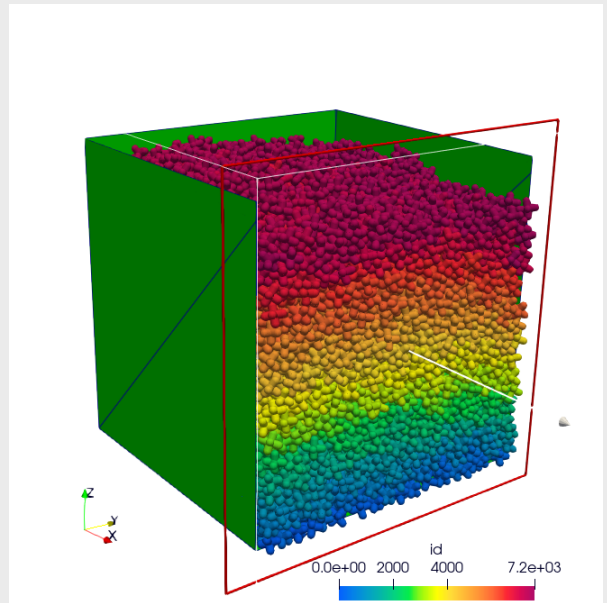
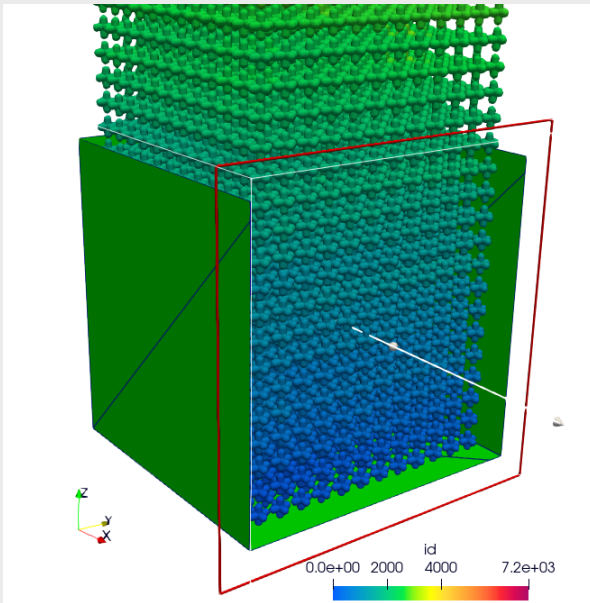
Note

These examples have minimal test cases (*_mini.msp*).

Polyhedra With a RShape Driver

Start

End



3.2. Examples Using Polyhedra

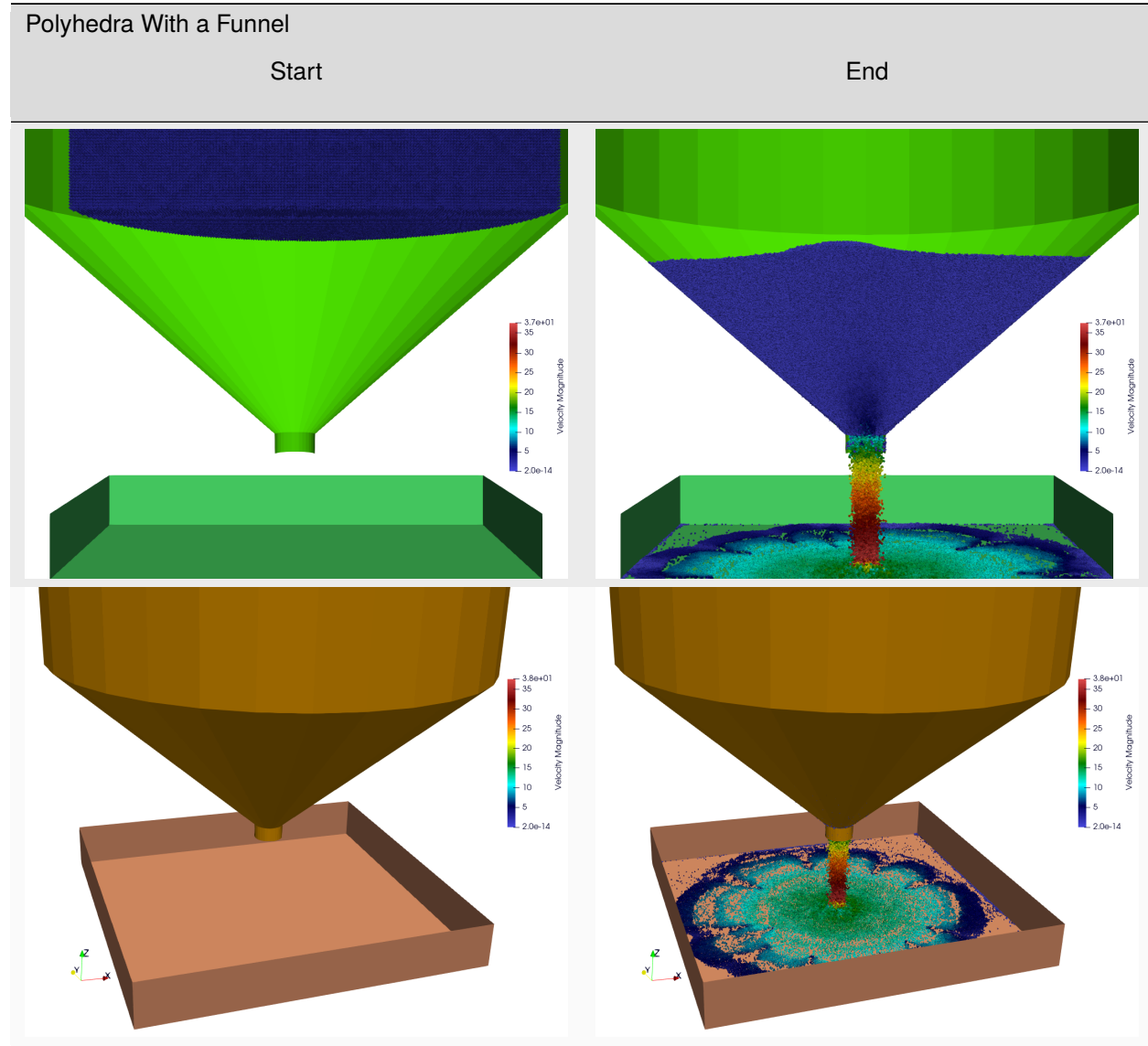
3.2.5 Example 5: Funnel

This simulation example illustrates the gravitational drop of a set of 1.3 million hexapods into a funnel. The funnel is represented using a mesh of faces (RShape Driver).

- Input file: `example/polyhedra/funnel/funnel.msp`

Note

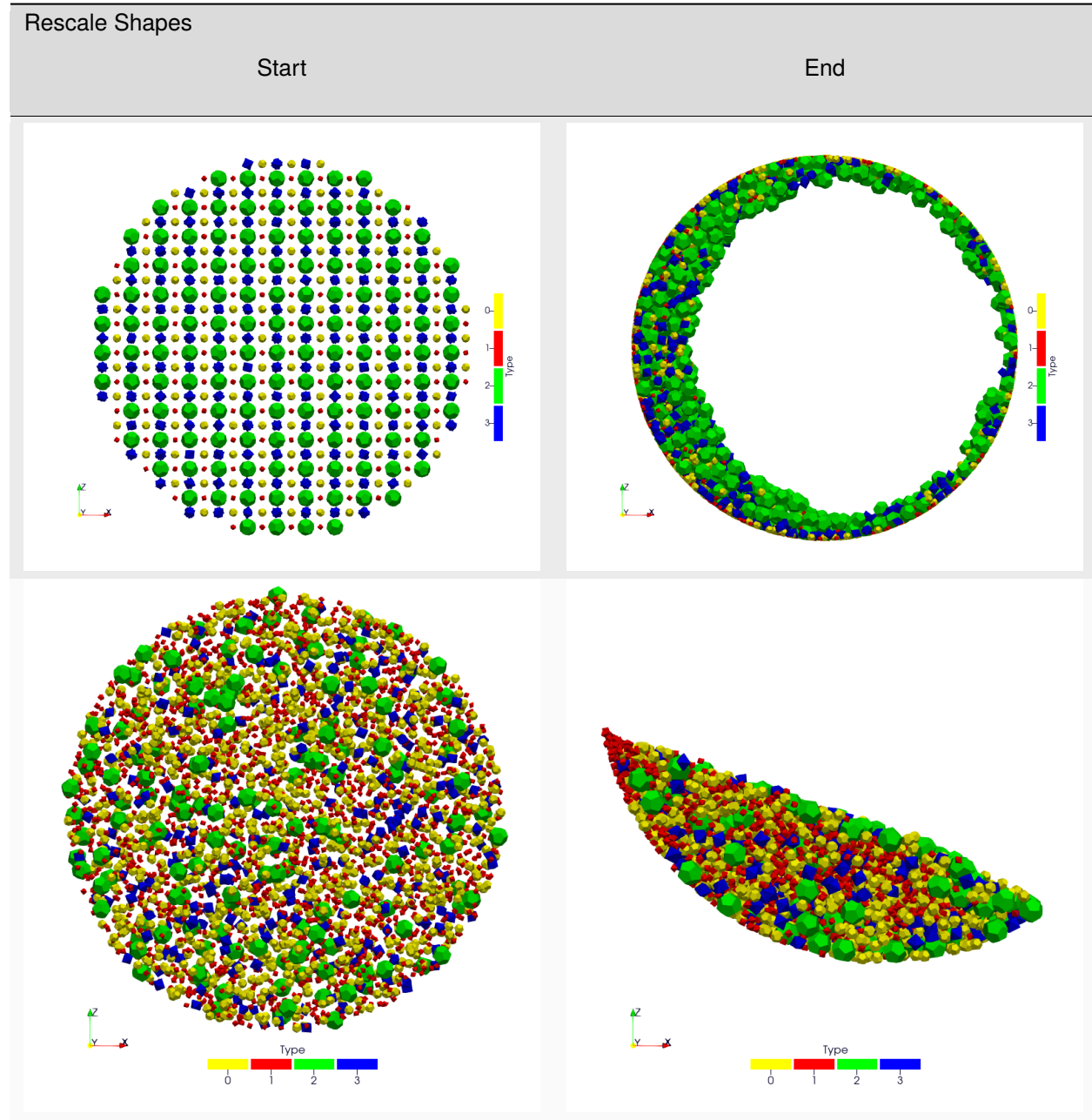
This example is designed to run on a supercomputer or recent GPUs, so don't run it on a laptop.



3.2.6 Example 6: Rescale shape

In this simulation, two shapes are selected: PolyR and Octahedron. Each shape is then duplicated with its size doubled, resulting in PolyRSize2 and OctahedronSize2. The setup is a rotating drum, where particles are generated using two different methods: FCC and RSA. The simulation runs for 1,000,000 time steps with a time step size of $dt = 5e-5$.

- Input files:
 - FCC: `example/polyhedra/rescale_shape/scale_2_shapes_fcc.msp`
 - * 2,144 polyhedra or 536 polyhedra of each type
 - * 1,000,000 timesteps
 - * 1 hour and 23 minutes on 12 cores (OpenMP)
 - RSA: `example/polyhedra/rescale_shape/scale_2_shapes_rsa.msp`
 - * 3,698 polyhedra, the volume fraction used for every type is 0.055
 - * 1,000,000 timesteps
 - * 38 minutes on 12 cores (OpenMP)

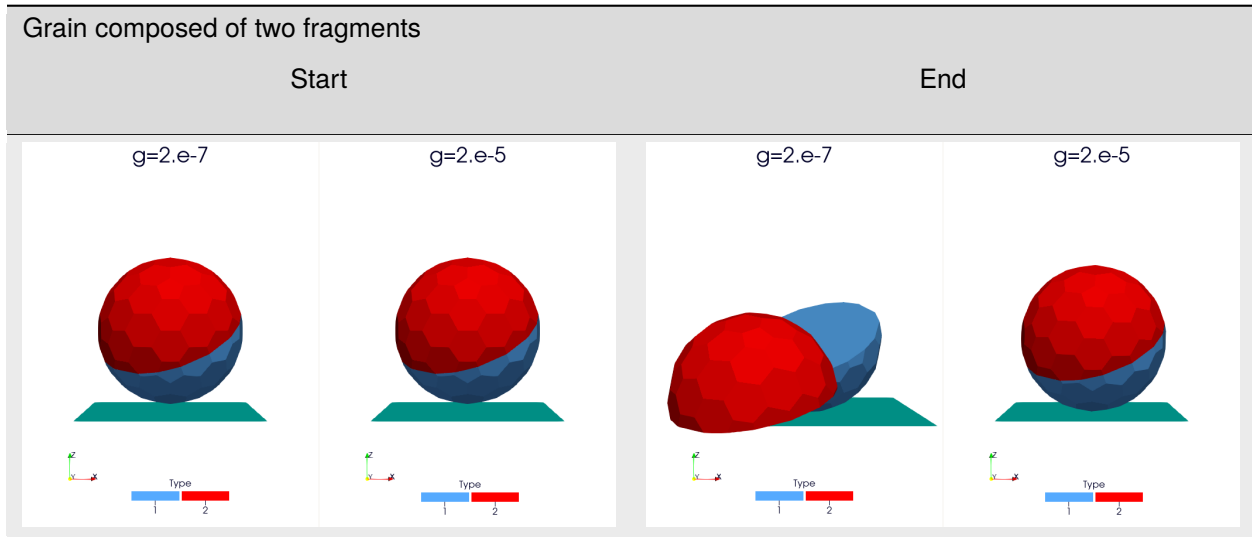


3.2.7 Example 7: Fragmentation with two fragments

This simulation demonstrates the fragmentation capability. The test case considers a grain composed of two fragments that is rapidly projected onto a rigid surface. Two configurations are compared. In the first one, the grain remains intact because the breakage criterion, governed by the parameter g , is not reached. In the second configuration, a lower value of g leads to the failure of the interface between the two fragments, resulting in fragmentation.

- Input files:
 - With Crackling: `example/polyhedra/fragmentation/two_fragments_break.msp`
 - * 2 fragments
 - * 1,800,000 timesteps

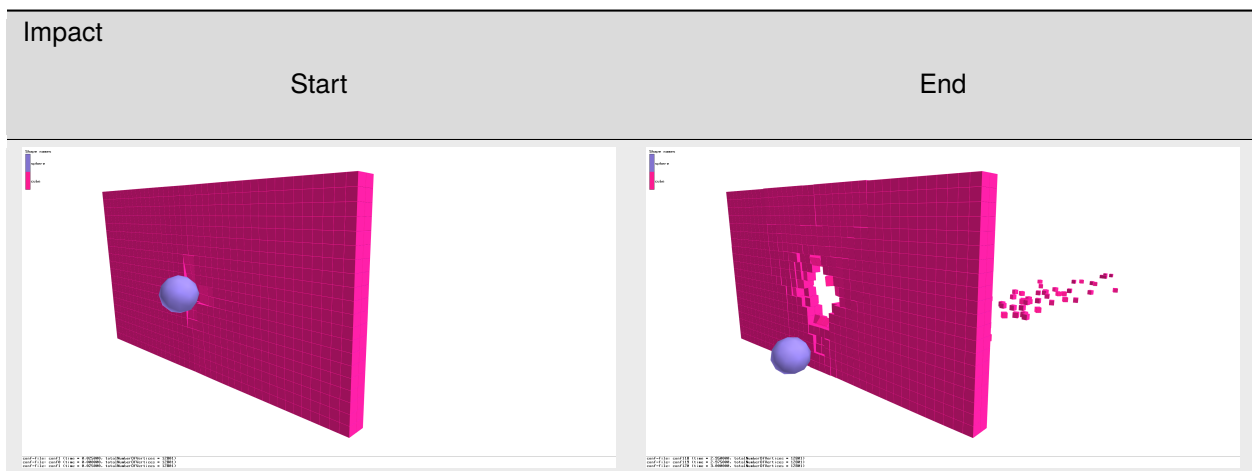
- * 84 seconds on 1 core
- No Crackling: `example/polyhedra/fragmentation/two_fragments_unbreak.msp`
 - * 2 fragments
 - * 1,800,000 timesteps
 - * 68 seconds on 1 core



3.2.8 Example 8: Impact of a Sphere

This simulation reproduces the impact of a high-speed projectile on a wall composed of cubic particles. These particles are linked by cohesive interactions (springs) between the vertices of their faces. The wall is placed on a surface with a high friction coefficient to prevent it from shifting. This simulation also incorporates a periodic boundary condition along the direction of the wall.

- Input files: `exaDEM/example/polyhedra/basic_shape/impact_with_frag.msp`
 - 1,601 particles
 - 300,000 timesteps
 - 445 seconds on 12 cores



Front view:

Back view:

3.3 Show-cases

3.3.1 On a laptop or single node

3.3.1.1 Rotating drum simulation

Simulation of 48,000 spheres on a rotating drum after reading an .stl file. This simulation takes around one hour and 10 minutes on a laptop with 12 cores with OpenMP. The input files are available here: `example/spheres/cylinder_stl/cylinder_stl.msp`

Simulation of 140,000 spheres falling into a cell. A blade is then activated with rotational and translational motion. This simulation takes around 4-5 days on a single node of 32 cores. A tutorial is available with polyhedra in the tutorial section.

TODO: Add input files

3.3.2 On a supercomputer

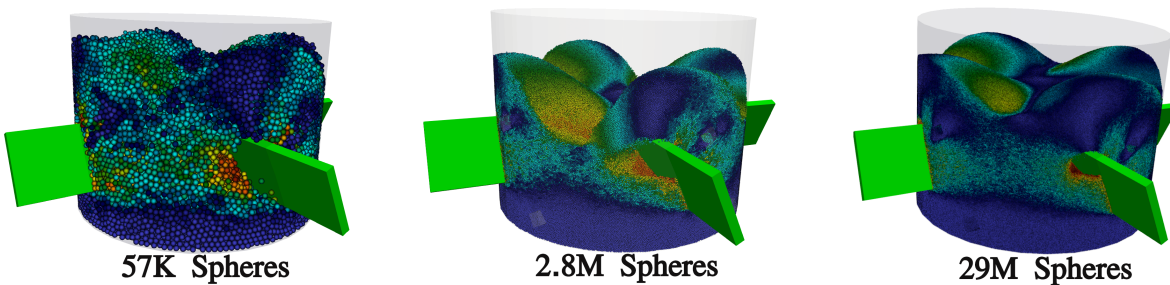
3.3.2.1 Mixer simulation

Simulation of a mixer into which spherical particles are falling. The simulation consists of rotating a propeller at 0.25 rad/s while using progressively smaller grains. This simulation is based on an example provided in the `chronoDEM::gpu` code and described in the following paper: “Chrono::GPU: An open-source simulation package for granular dynamics using the discrete element method” [6]. To accelerate simulation time, the time step was increased to 1×10^3 , and the contact law parameters were adjusted accordingly.

The input files for the different scenarios are available in: `exaDEM/example/spheres/mixer`

- `mixer_57k_sph.msp`
- `mixer_3M_sph.msp`
- `mixer_29M_sph.msp`

Below is an image showing the different scenarios based on the number of particles:



4.1 Spherical Particle

In this section, we will describe the various information used to build simulations with spherical particles. Note that `exaDEM::Interaction` and `Shape` are described in the section: `Polyhedron`.

4.1.1 Spheres - Interaction / Contact

The `exaDEM::Interaction` class in `ExaDEM` is used to model various types of interactions between spheres themselves and between spheres and `drivers`. This class serves as a crucial component for identifying two elements within the data grid and characterizing the type of interaction between them. Note that, for the sake of consistency between the sphere and sphere sections, we use the same interaction structure, and that the `type` field that qualifies the nature of the interaction has the same values as for polyhedra.

Interaction Class Attributes:

- id_i and id_j : Id of both spheres.
- $cell_i$ and $cell_j$: Indices of the cells containing the interacting spheres.
- p_i and p_j : Positions of the spheres within their respective cells.
- sub_i and sub_j : Indices of the vertex, edge, or face of the sphere involved in the interaction.
- `type`: Type of interaction (integer). See Interaction Glossary.
- `friction` and `moment`: Storage used for temporary computations.

Note

When the interaction involves a sphere and a `driver`, particle `j` is used to locate the `driver`. In this scenario, `cell_j` represents the index of the `driver`. If the `driver` utilizes a shape, such as with `RShapeDriver`, `sub_j` is also utilized to store the index of the vertex, edge, or face.

Table 1: Glossary of Interaction types for Spheres

Value	Type	Description
0:	Sphere - Sphere	Contact between two vertices of two different spheres
4	Sphere - Cylinder	Contact between a vertex of a sphere and a cylinder
5	Sphere - Surface	Contact between a vertex of a sphere and a rigid surface or wall
6	Sphere - Ball	Contact between a vertex of a sphere and a ball / sphere
7	Sphere - Vertex (Driver)	Contact between a vertex of a sphere and a vertex of a <code>RShape Driver</code>
8	Sphere - Edge (Driver)	Contact between a vertex of a sphere and an edge of a <code>RShape Driver</code>
9	Sphere - Face (Driver)	Contact between a vertex of a sphere and a face of a <code>RShape Driver</code>

Interaction Class Usage With Spheres:

To retrieve data associated with a specific interaction between two spheres, the attributes of the `exaDEM::Interaction` class are used to identify cells, positions, and interaction types. This information is then utilized within simulation computations to accurately model interactions between spheres, considering the interaction type.

These interactions are utilized as a level of granularity for intra-node parallelization, applicable to both CPU and upcoming GPU implementations. The interactions are populated within the `nbh_sphere` operator and subsequently processed in the `contact_sphere` operator.

In summary, the `exaDEM::Interaction` class provides a crucial data structure for managing interactions between spheres and drivers within DEM simulations. By storing information such as cell numbers, positions, and interaction types, it enables precise modeling of physical interactions between simulated objects.

4.1.2 Operators

4.1.2.1 `randomize_radius`

The `randomize_radius` operator randomly modifies the radius of spherical particles and updates their physical properties accordingly.

For each particle, a new radius is sampled from a Gaussian distribution centered on the current radius. The width of the distribution is controlled by the `relative_deviation` parameter. After the new radius is computed, the particle mass and inertia tensor are updated consistently assuming that the particle density remains constant.

The radius variation is bounded in order to avoid unrealistic values.

Parameters

- `relative_deviation` (double, required)
Relative deviation applied to the particle radius. For example, `0.2` corresponds to a variation of approximately $\pm 20\%$ around the current radius.
- `allow_exceed` (bool, optional, default = true)
Controls whether the new radius is allowed to exceed the original radius.
 - `true`: the radius may increase or decrease within the allowed range.
 - `false`: the radius cannot become larger than the initial radius.
- `region` (optional)
A `ParticleRegionCSG` definition used to restrict the operation to particles belonging to a specific region.
- `particle_regions` (optional)
Collection of particle regions used to evaluate the `region` expression.

Example

Basic usage:

```
- randomize_radius:  
  relative_deviation: 0.2
```

Allowing the radius to increase beyond its initial value:

```
- randomize_radius:  
  relative_deviation: 0.2  
  allow_exceed: true
```

Applying the operator only to a specific particle region:


```
- randomize_radius:
  relative_deviation: 0.15
  allow_exceed: true
  region: my_region
```

4.2 R-Shape / Polyhedron

In this section, we will describe the various information used to build simulations with R-shape or sphero-polyhedron particles.

4.2.1 Overview

The polyhedra implemented in ExaDEM are sphero-polyhedra, i.e. the vertices of the polyhedra are considered as spheres and the edges as cylinders. To achieve this, ExaDEM incorporates many of the features of the Rockable DEM code developed at CNRS (<https://github.com/richefeu/rockable>, <https://richefeu.github.io/rockable/quickStart.html>). ExaDEM relies in particular on a Shape class containing information about the polyhedron (vertex, edge, face, and Minkowski radius) and an interaction class used to qualify a contact between polyhedra. It's important to note that the spheropolyhedron approach can be used to simulate complex non-convex particles such as hexapods.

4.2.2 Shape

The Shape class provides all the information on vertices, edges, and faces, but it also provides other support to speed up calculations, such as OBB sets for each type of information. ExaDEM provides many features linked to the Shape class, such as reading .shp files (the format used by Rockable), as well as other functions such as outputting a .vtk file of the shape. This class is defined by the following properties:

Table 2: Shape Class Properties

Property Name	Attribute Name	Description
Vertices	m_vertices	List of the polyhedron's vertices.
Edges	m_edges	List of the polyhedron's edges.
Faces	m_faces	List of the polyhedron's faces.
Minkowski Radius	m_radius	Minkowski radius (radius of the vertices).
Volume	m_volume	Total volume of the polyhedron.
Inertia Coefficient	m_inertia_on_mass	Inertia coefficient normalized by mass.
Name	m_name	Name of the object (defaults to "undefined").
OBB	obb	Oriented Bounding Box of the polyhedron.
Vertex OBBs	m_obb_vertices	List of OBB for each vertex (only for Big Shape).
Edge OBBs	m_obb_edges	List of OBB for each edge.
Face OBBs	m_obb_faces	List of OBB for each face.

Note

OBB (Oriented Bounded Boxes) are enlarged of the Minskowki radius.

Note

By default, every shape is stored in a list of shapes, and the maximum cut-off radius is deduced from these shapes. Note that a cut-off radius that is too large can drastically reduce simulation performance. That's why, do not put

big shapes using the classical way (i.e. `read_shape_file`), big shapes should be defined as drivers.

Shape example (octahedron, 6 vertices, 12 edges, and 8 faces):

```
<
name Octahedron
radius 0.1
preCompDone y
nv 6
0.2310789034541148 -0.2310789034541148 0.0
0.2310789034541148 0.2310789034541148 0.0
0.0 0.0 0.32679491924311227
-0.2310789034541148 -0.2310789034541148 0.0
-0.2310789034541148 0.2310789034541148 0.0
0.0 0.0 -0.32679491924311227
ne 12
0 1
2 1
2 0
0 3
2 3
3 4
4 2
4 1
5 0
5 1
5 4
5 3
nf 8
3 0 1 2
3 2 3 4
3 1 2 4
3 0 2 3
3 0 5 1
3 0 5 3
3 3 5 4
3 4 5 1
obb.extent 0.33107890345411484 0.33107890345411484 0.4267949192431123
obb.e1 1.0 0.0 0.0
obb.e2 0.0 1.0 0.0
obb.e3 0.0 0.0 1.0
obb.center 0.0 0.0 0.0
position 0.0 0.0 0.0
orientation 1.0 0.0 0.0 0.0
volume 0.16666666666666666
I/m 0.04999999999999999 0.04999999999999999 0.04999999999999999
>
```

Or a sphere (1 vertex, 0 edge, 0 face):

```
<
name alpha1
```

(continues on next page)

(continued from previous page)

```

radius 0.5
preCompDone y
nv 1
0 0 0
ne 0
nf 0
obb.extent 0.5 0.5 0.5
obb.e1 1 0 0
obb.e2 0 1 0
obb.e3 0 0 1
obb.center 0 0 0
volume 0.523598775598299
I/m 0.1 0.1 0.1
>

```

It's important to note that using a shape of a spherical particle with a polyhedron configuration instead of directly using a sphere configuration decreases overall performance due to unnecessary calculations, such as applying an orientation to a vertex. We have observed that in this case, simulations are about 2 to 3 times slower.

- Operator Name: `read_shape_file`
- Description: This operator initialize the shapes data structure from a shape input file.
- Parameters:
 - `filename`: Input file name (.shp)
 - `scale_factor`: rescale all shapes. Optional parameter.
 - `rename`: rename all shapes. Optional parameter

YAML example:

```

- read_shape_file:
  filename: shapes.shp
  rename: [PolyR, Octahedron]
- read_shape_file:
  filename: shapes.shp
  rename: [ PolyRSize2, OctahedronSize2]
  scale_factor: [ 2.0, 2.0]

```

Example: See *Example 6: Rescale shape* .

4.2.3 Basic Shapes

ExaDEM provides some basic shapes without using a shape file.

- Operator Name: `add_sphere`
- Description: Add a sphere to the shape lists.
- Parameters:
 - `length`: Define cube length.
 - `minskowski`: Define the Minskowski value.

```
- add_sphere:  
  name: MySphere  
  minskowski: 1.0
```

- Operator Name: add_cube
- Description: Add a cube to the shape lists.
- Parameters:
 - length: Define cube length.
 - name: Set Shape name. Default is “cube”.
 - minskowski: Define the Minskowski value.

YAML Example:

```
- add_cube:  
  name: MyCube  
  length: 0.5  
  minskowski: 0.25
```

- Operator Name: add_rice
- Description: Add a rice to the shape lists.
- Parameters:
 - length: Define rice length.
 - name: Set Shape name. Default is “rice”.
 - minskowski: Define the Minskowski value.

YAML Example:

```
- add_rice:  
  name: MyRice  
  length: 0.5  
  minskowski: 0.025
```

4.2.4 Polyhedra - Interaction / Contact

The `exaDEM::Interaction` class in ExaDEM is used to model various types of interactions between polyhedra and between polyhedra and drivers. This class serves as a crucial component for identifying two elements within the data grid and characterizing the type of interaction between them.

Interaction Class Attributes:

- id_i and id_j : Id of both polyhedra.
- $cell_i$ and $cell_j$: Indices of the cells containing the interacting polyhedra.
- p_i and p_j : Positions of the polyhedra within their respective cells.
- sub_i and sub_j : Indices of the vertex, edge, or face of the polyhedron involved in the interaction.
- type: Type of interaction (integer). See Interaction Glossary.
- friction and moment: Storage used for temporary computations.

Note

When the interaction involves a polyhedron and a driver, particle j is used to locate the driver. In this scenario, $cell_j$ represents the index of the driver. If the driver utilizes a shape, such as with `RShapeDriver`, sub_j is also utilized to store the index of the vertex, edge, or face.

Table 3: Glossary of Interaction types

Value	Type	Description
0	Vertex - Vertex	Contact between two vertices of two different polyhedra
1	Vertex - Edge	Contact between a vertex and an edge of two different polyhedra
2	Vertex - Face	Contact between a vertex and a face of two different polyhedra
3	Edge - Edge	Contact between two edges of two different polyhedra
4	Vertex - Cylinder	Contact between a vertex of a polyhedron and a cylinder
5	Vertex - Surface	Contact between a vertex of a polyhedron and a rigid surface or wall
6	Vertex - Ball	Contact between a vertex of a polyhedron and a ball / sphere
7	Vertex - Vertex (Driver)	Contact between a vertex of a polyhedron and a vertex of a RShape Driver
8	Vertex - Edge (Driver)	Contact between a vertex of a polyhedron and an edge of a RShape Driver
9	Vertex - Face (Driver)	Contact between a vertex of a polyhedron and a face of a RShape Driver
10	Edge - Edge (Driver)	Contact between an edge of a polyhedron and a edge of a RShape Driver
11	Vertex (Driver) - Edge	Contact between a vertex of a RShape Driver and an edge of a polyhedron
12	Vertex (Driver) - Face	Contact between a vertex of a RShape Driver and a face of a polyhedron
13	Inner Bond	Contact within a grain composed of fragments (polyhedra)

Interaction Class Usage:

To retrieve data associated with a specific interaction between two polyhedra, the attributes of the `exaDEM::Interaction` class are used to identify cells, positions, and interaction types. This information is then used within simulation computations to accurately model interactions between polyhedra, considering the interaction type.

These interactions are used as a level of granularity for intra-node parallelization, applicable to both CPU and upcoming GPU implementations. The interactions are populated within the `nbh_polyhedron` operator and subsequently processed in the `contact_polyhedron` operator.

In summary, the `exaDEM::Interaction` class provides a crucial data structure for managing interactions between polyhedra and drivers within DEM simulations. By storing information such as cell numbers, positions, and interaction types, it enables precise modeling of physical interactions between simulated objects.

Grid Of Interactions:

In ExaDEM, interactions are stored in the form of a grid of cells (AOSOA), the cell (SOA) then containing a `GridExtraDynamicDataStorageT`, i.e. a data structure similar to a vector of `Interactions` + particle information vector. This data structure facilitates the migration of information between MPI processes when the interaction is considered to be always active (i.e. the two polyhedra are always in contact from one time step to the next). For more details in code, see `src/polyhedra/include/exaDEM/interaction/grid_cell_interaction.hpp` and the `extra_storage` package in ExaNBody.

Classifier:

To improve the implementation of kernels linked to GPU interactions, `exaDEM` relies on the `classifier` class, which sorts all interactions by type in an SOA, so that several kernels can be launched, each dealing with the same type of interaction. The aim is to limit instruction divergence between GPU threads.

It's important to point out that this data structure complements the interaction grid. The main idea is to classify and unclassify interaction information as long as the data has not changed (`cell migration`, `move particle`, `IO`). To

achieve this, we use two operators: `classify` and `unclassify`.

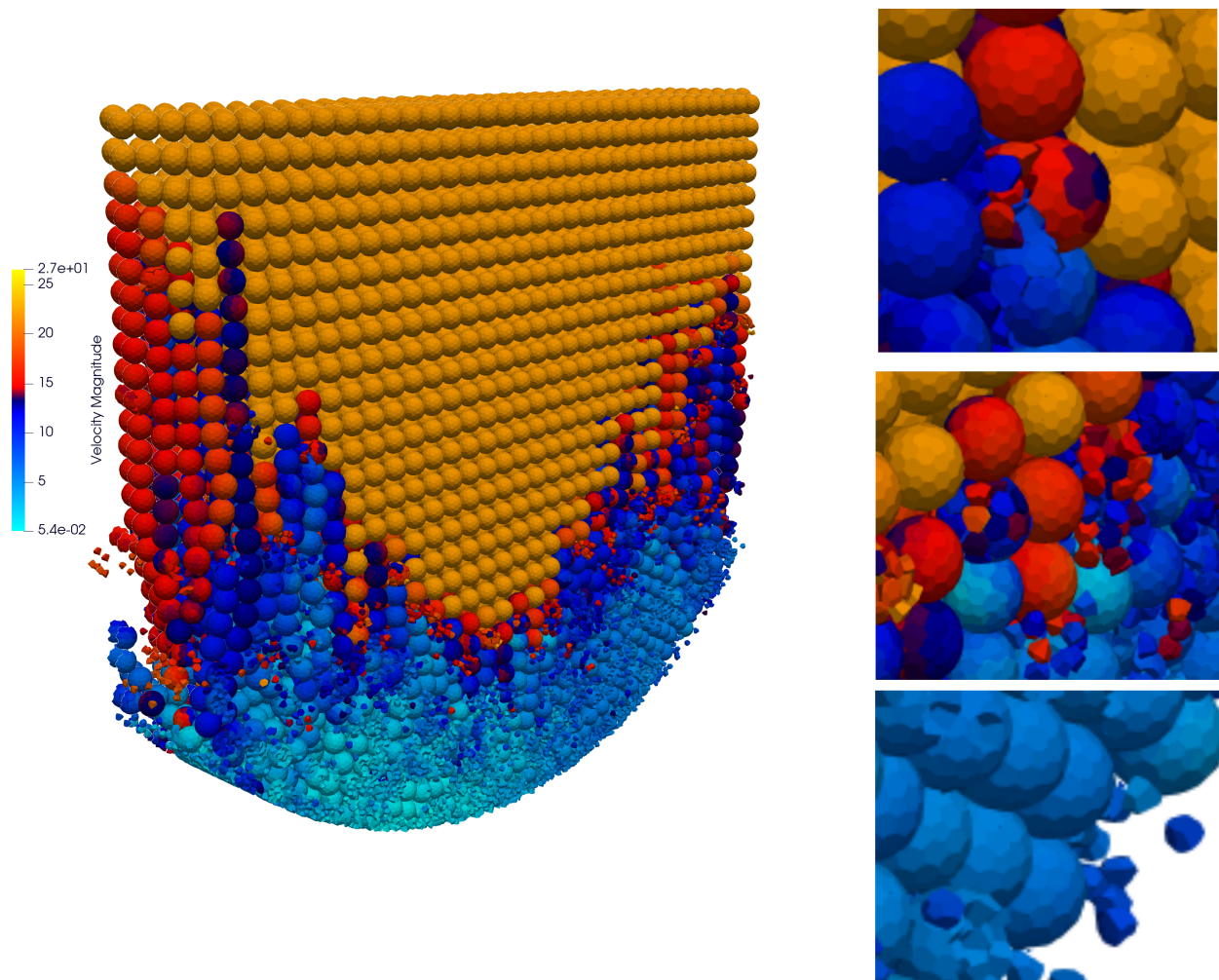
Using the classifier is currently the default strategy in `exaDEM` for spheres and polyhedra.

4.2.5 Fragmentation Feature

Note

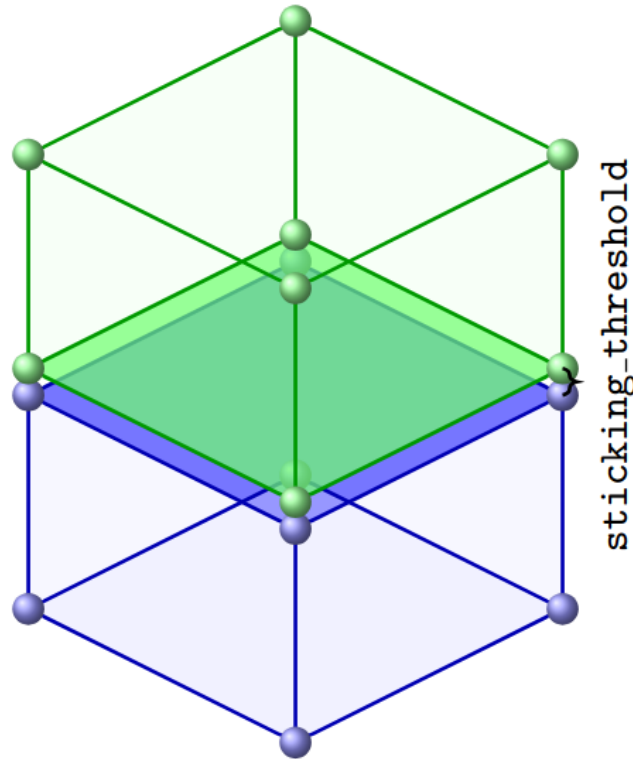
This feature is currently **experimental**.

The strategy for handling fragmentation in `exaDEM` consists of pre-cutting the grains into small polyhedrons and adding springs between the opposite vertices for the faces to be bonded. To use these developments, you must include the `config_fragmentation.msp` file instead of `config_polyhedra.msp`.



The criterion for sticking particles depends on the distance between opposite vertices based on the distance defined by: `sticking_threshold: 1.e-04` to be defined in the global operator, please also define `apply_particle_sticking: true`.

Finally, `exaDEM` will process interfaces that are a set of `InnerBond` interactions (`typeId = 13`) and check at each time



step whether the energy released exceeds a certain threshold (depending on the surface area and a parameter g).

If an **Interface** is broken, the interactions are removed and the interaction lists are reconstructed. Note that if two particles are stuck together by an interface, no other interactions (vertex-vertex, vertex-edge, etc.) are possible.

4.2.6 Data layout: Particle Vertices

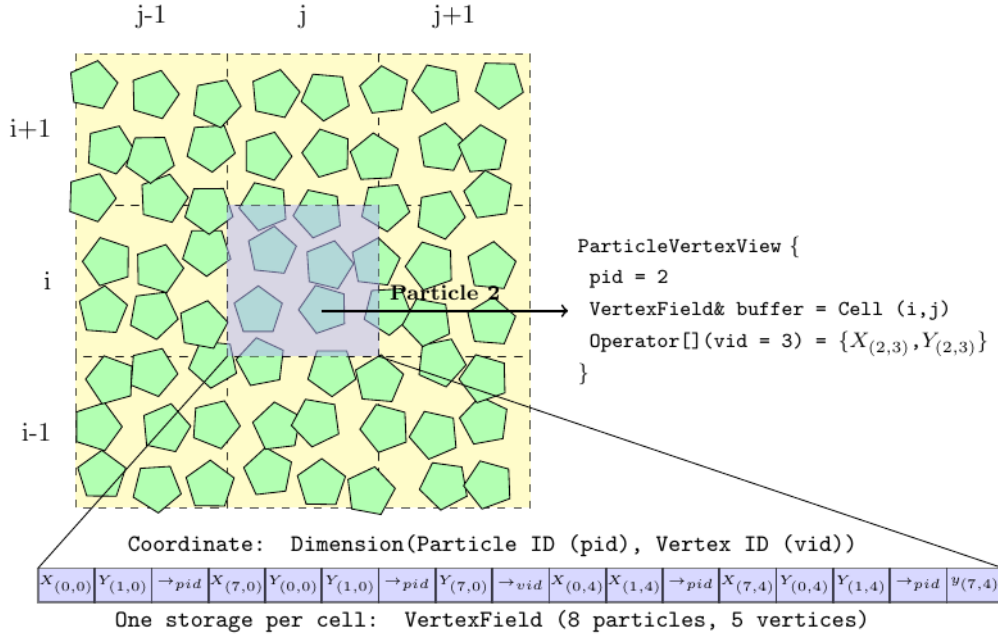
The vertices of the polyhedra are stored in a different grid structure called **CellVertexField**. It is composed as a grid of **VertexFields** and is reallocated by the `compute_vertices` operator. The following image illustrates the memory layout of the vertices:

- Operator name: `compute_vertices`
- Description: This operator computes the vertices for every polyhedron.
- Parameters:
 - `resize_vertex`: enable to resize the data storage used for vertices, default is `true`
 - `minimize_memory_footprint` enable to resize the data storage using only the maximum of vertices according to the particle shapes into a cell. This option is useful if there are some particles with a very high number of particles, default is `false`.

YAML examples:

```
compute_new_vertices:
- compute_vertices:
    resize_vertex: true
compute_fast_vertices:
```

(continues on next page)



(continued from previous page)

```
- compute_vertices:
  resize_vertex: false
```

4.3 Particle Fields

In this section, we provide an overview of ExaDEM operators associated with field mutation. This section is divided into three subsections: the unified field operator for spheres and polyhedra, field operators applicable to particles of any type, and field operators specifically designed for polyhedra.

4.3.1 Unified Field Operators

Warning

This operator has many possible combinations, some of which may be wrong. Please report any doubts.

- Operator name: `set_fields`:
- **Parameters REQUIRED:**
 - [bool] *polyhedra*: Define if the kind of particles is polyhedron or sphere.
 - [std::vector<string>] *type*: Particle type names.
- **Parameters OPTIONAL:**
 - [std::vector<double>] *density*: List of density values. If not defined, density is 1.
 - [std::vector<double>] *radius*: List of radius values. If not defined, radius is 0.5 for spheres, do not define it for polyhedra.
 - [std::vector<Vec3d>] *velocity*: List of velocity values. If not defined, velocity is [0,0,0].

- [std::vector<double>] *sigma_velocity*: Standard deviation (sigma). If not defined, the normal distribution is not applied.
- [std::vector<Vec3d>] *angular_velocity*: List of angular velocity values. If not defined, angular velocity is [0,0,0].
- [std::vector<double>] *sigma_angular_velocity*: Standard deviation (sigma). If not defined, the normal distribution is not applied.
- [std::vector<Quaternion>] *quaternion*: List of orientations. If not defined, quaternion is [w = 1,0,0,0].
- [std::vector<bool>] *random_quaternion*: Choice if the orientation is random or not. If not defined, random is false.
- [std::vector<uint32_t>] *group*: Group index associated to each particle type (same order as *type*). If not defined, group is 0 for all particles.

Note

For spheres, you need to call `particle_type` operator to create the particle map required by `set_fields`.

Note

Since `exaDEM-1.2.3`, the `group` field is used by the multi-material contact operators (`multimat_contact_params`, `drivers_contact_params`, `inner_bond_params`) to select which set of contact parameters applies between two particles. Before 1.2.3, the particle type itself was directly used to look up these contact parameters. With `group`, several particle types can share the same group and therefore reuse the same contact parameters, decoupling the (shape-related) `type` from the (contact-law-related) `group`. See the Multi-Material section of the Force Field page for details. Groups can also be (re)assigned after initialization with the `set_group` operator.

YAML example (Spheres):

```
- particle_type:
  type: [ Sphere1, Sphere2 ]
- lattice:
  structure: BCC
  types: [ Sphere1, Sphere2 ]
  size: [ 1.0 , 1.0 , 1.0 ]
- set_fields:
  polyhedra: false
  type: [ Sphere1, Sphere2 ]
  group: [ 0, 1 ]
  radius: [ 0.5, 0.25 ]
  density: [ 0.02, 0.01 ]
  velocity: [ [0,0,0], [0,0,0] ]
  sigma_velocity: [ 0.01, 0.01 ]
```

YAML example (Polyhedra):

```
- read_shape_file:
  filename: alpha3.shp
- read_shape_file:
  filename: octahedron.shp
```

(continues on next page)

(continued from previous page)

```

- lattice:
  structure: BCC
  types: [alpha3, Octahedron ]
  size: [ 1.5 , 1.5 , 1.5 ]
  repeats: [ 30 , 30 , 30 ]
  enlarge_bounds: 0.0 m
- set_fields:
  polyhedra: true
  type: [ alpha3, Octahedron ]
  group: [ 0, 1 ]
  velocity: [ [0,0,0], [0,0,0] ]
  sigma_velocity: [ 0.1, 0.1 ]
  random_quaternion: [ true, true ]

```

4.3.2 Field Operators For All Particles

This repertory plugin only provides operators for modifying fields, especially at initialization. The following operators are based on the functor *set* and initialize one or more fields:

- **set_densities_multiple_materials:**
 - [std::vector<double>] *densities*
 - Comment: mass is deduced from the density and radius
- **set_density:**
 - [double] *density*
- **set_homothety:**
 - [double] *homothety*
- **set_type:**
 - [uint32_t] *type*
- **set_material_properties:**
 - [uint8_t] *type*
 - [double] *rad*
 - [double] *density*
 - [Quaternion] *quat*
- **set_quaternion:**
 - [Quaternion] *quat*
 - [bool] *random*, default is false.
- **set_radius:**
 - [double] *rad*
- **set_radius_multiple_materials:**
 - [std::vector<double>]` radius` (list of radii according to types)
- **set_rand_vrot_arot:**
 - [double] *var_vrot* (variance), default = 0

- [double] *var_arot* (variance), default = 0
- [Vec3d] *mean_arot* (mean), default = {0,0,0}
- [Vec3d] *mean_vrot* (mean), default = {0,0,0}
- Comment : This operator sets the angular acceleration and velocity using a normal distribution
- **set_rand_velocity:**
 - [double] *var* (variance), default = 0
 - [Vec3d] *mean*, default = {0,0,0}
- **update_inertia**
- **set_group (since exaDEM-1.2.3):**
 - [std::vector<string>] *type* REQUIRED: List of particle type names.
 - [std::vector<uint32_t>] *group* REQUIRED: Group index associated to each type (same order as *type*).
 - [uint32_t] *n_groups* OUTPUT: Number of distinct groups (max group index + 1), written back for downstream operators such as *multimat_contact_params*, *drivers_contact_params* and *check_group_completeness*.
 - Comment: Assigns the *group* field to every particle according to its type. It can be used to (re)define groups after *set_fields* has been called, without having to redefine every other field.

Note

It is possible to specify the region slot to apply the following kernel to a special spatial area: *set_homothety*, *set_type*, *set_quaternion*, *set_rand_velocity*, *set_material_properties*, *set_radius*, and *update_inertia*.

YAML example:

```
- set_radius:
  rad: 0.5
- set_quaternion
- set_rand_velocity:
  var: 0.1
  mean: [0.0,0.0,0.0]
- set_density:
  density: 0.02
- set_rand_vrot_arot
```

set_group YAML example:

```
init_polyhedra:
- set_group:
  type: [ alpha3, Octahedron, Cube ]
  group: [ 0, 1, 0 ]

init_spheres:
- set_group:
  type: [ Sphere1, Sphere2, Sphere3 ]
  group: [ 0, 1, 0 ]
```

Explore a minimal example provided in the “tutorial” section to understand how you can add your own mutator_field operator.

4.3.3 Field Operators For Polyhedra

In this section, we briefly describe field mutator operators that relate to data contained within the shape data structure (see the Polyhedra Section for more details).

- **density_from_shape**
[This operator deduces the particle mass from the shape volume and the particle density.]
 - [double] *density*
- **inertia_from_shape** : This operator deduces the particle inertia from the shape constant I/M and the particle mass.
- **radius_from_shape** : This operator computes the maximum radius cutoff in function of shape types and stores the radius cutoff for every particle corresponding to their shape types.

Note

It is possible to specify the **region** slot to apply the following kernel to a special spatial area: **radius_from_shape**.

4.4 Force Field

The force field encompasses a broader set of operators and mechanisms responsible for computing forces acting on particles within the particle environment. It includes various types of forces such as gravitational forces, contact forces, and other external influences that affect particle dynamics.

4.4.1 Contact Force Laws

Contact's Law in the context of the Discrete Element Method (DEM) refers to the principle used to calculate forces between particles based on their relative displacements. In DEM simulations, Contact's Law is applied to model interactions between particles, enabling the simulation of elastic deformation and linear force behaviors within particle-based systems.

We distinguish two kinds of interaction :

- pure contact interaction
- cohesive interaction

Here after are given the main laws available in exaDEM :

Name	type	Description
hooke	pure contact	Default configuration : elastic linear normal force, normal viscosity force, Coulomb friction tangent force and rolling resistance
cohesiv	Cohe-sive	Addition of a normal cohesive force depending on the distance between particles
dmt	Cohe-sive	Addition of an adhesive law at contact to model Van der Waals force for hard particles

The variables required to describe interactions are:

Variable	Description
$\backslash(\text{cp})\backslash$	The contact position
$\backslash(\text{r}_i)\backslash$	The position of the particle i
$\backslash(\text{r}_j)\backslash$	The position of the particle j
$\backslash(\text{v}_i)\backslash$	The velocity of the particle i
$\backslash(\text{v}_j)\backslash$	The velocity of the particle j
$\backslash(\text{vrot}_i)\backslash$	The angular velocity of the particle i
$\backslash(\text{vrot}_j)\backslash$	The angular velocity of the particle j
$\backslash(\text{m}_i)\backslash$	The mass of the particle i
$\backslash(\text{m}_j)\backslash$	The mass of the particle j
$\backslash(\text{delta}_n)\backslash$	The interpenetration / particle overlap

Each kind of interaction also requires additional constants, reflecting for most of them mechanical properties:

Constant	Description
$\backslash(\Delta_t)\backslash$	The timestep increment
$\backslash(\alpha_n)\backslash$	The damping rate
$\backslash(k_n)\backslash$	The normal stiffness coefficient
$\backslash(k_t)\backslash$	The tangential stiffness coefficient
$\backslash(v_t)\backslash$	The relative tangential velocity
$\backslash(k_r)\backslash$	The rotational stiffness coefficient
$\backslash(\mu)\backslash$	The coefficient of friction

4.4.1.1 Elastic linear force with friction tangent force (hooke)

In the Discrete Element Method (DEM), the equations of motion (translations and rotations) are discretized in time. Only the rigid-body displacements are considered. Small overlaps between the particles are allowed and used as strain variables. The total contact force between particle i and particle j is given by

$$\mathbf{f}_{ij} = f_n \mathbf{n} + \mathbf{f}_t$$

where f_n the normal component of the contact force and $\mathbf{f}_t$ is the tangential force vector. These forces are expressed in the local contact frame $(\mathbf{n}, \mathbf{t}, \mathbf{s})$ as a function of the overlaps and tangential displacements. They are calculated from force laws that generally describe frictional contact interactions. An important feature of DEM is to allow the particles to overlap. This overlap δ_n represents a normal strain localized in the vicinity of the contact point. A simple linear relation is assumed between normal contact force and δ_n . This is consistent with the fact that the overlaps allow for a penalty-based explicit formulation of particle motions, i.e., the elastic repulsion force is mobilized to prevent penalizing the overlap. The condition of particle undeformability implies that the overlaps must stay small compared to particle size. In this linear approximation, the normal component of the contact force is given by

$$f_n = -k_n \delta_n + \nu_n v_n$$

where k_n is the normal stiffness coefficient, v_n the normal component of the relative velocity (between particle i and particle j), and ν_n is the viscous damping coefficient. ν_n is related to the restitution coefficient e_n by

$$\nu_n = \alpha_n \sqrt{2m_{\text{eff}} v_n}$$

$$\alpha_n = \frac{-\ln e_n}{\sqrt{\ln^2 e_n + \pi^2}}$$

where:

- ν_n is the viscous damping rate during the collision

- α_n is the damping parameter, calculated from the restitution coefficient e_n
- m_{eff} is the effective mass of the two colliding particles, defined by $m_{\text{eff}} = \frac{m_i m_j}{m_i + m_j}$

Note

The formulas are identical to those used in Rockable (see [Rockable Force Laws](#)) but are implemented differently to align with the *exaDEM* data structure.

The tangential force represents the frictional resistance between particles when they slide against each other. This force is calculated based on the relative tangential velocity (v_t) and a tangential stiffness parameter (k_t , keyword `ktContact`).

The **Coulomb friction model** is used to limit the tangential force, ensuring that it does not exceed the product of the friction coefficient μ and the normal force (f_n):

$$f_t \leq \mu \cdot f_n$$

The tangential force f_t is incrementally updated at each time step according to the following relation, starting from the onset of contact:

$$f_t = f_t + k_t \cdot v_t \cdot \Delta t$$

where: - f_t is the tangential force, - k_t is the tangential stiffness, - v_t is the relative tangential velocity, and - δt is the time step.

The friction force is reset to zero as soon as contact is lost.

4.4.1.2 cohesive normal law

This interaction requires two more parameters that are :

Constant	Description
$\backslash(\text{fc})\backslash$	Cohesive force threshold
$\backslash(\text{dncut})\backslash$	Distance cutoff for cohesive interaction

Three possibilities depending on the value of the interpenetration between $\backslash(\delta_n)\backslash$ two particles:

- $\backslash(\delta_n < -\text{dncut})\backslash$
- $\backslash(-\text{dncut} < \delta_n < 0)\backslash$
- $\backslash(0 < \delta_n < \text{dncut})\backslash$

Warning

Cohesive forces (*dncut*) are only applied if you use the operators `contact_*_*_*_cohesive`. Otherwise we only consider the case $d_n < 0.0$.

Formula between particle i and particle j if $\backslash(\delta_n < -\text{dncut})\backslash$:

$$\mathbf{f}_{ij} = -k_n \cdot \delta_n + \alpha_n \sqrt{2 \cdot m_{\text{eff}} v_n} + k_t \cdot v_t \cdot \Delta t$$

with the effective mass:

$$m_{\text{eff}} = \frac{m_i \cdot m_j}{m_i + m_j}$$

and the relative velocity norm:

$$v_n = (v_i - (cp - r_i) \wedge vrot_i) - (v_j - (cp - r_j) \wedge vrot_j)$$

Formula between particle i and particle j if $(-\delta_{cut} < \delta_n < 0)$:

$$\mathbf{f}_{ij} = f_n + k_t \cdot v_t \cdot \Delta_t$$

with:

$$f_n = \begin{cases} -k_n \cdot \delta_n + \alpha_n \sqrt{2 \cdot m_{eff}} v_n & \text{if } -k_n \cdot \delta_n + \alpha_n \sqrt{2 \cdot m_{eff}} v_n \geq -f_c \\ -f_c & \text{if } -k_n \cdot \delta_n + \alpha_n \sqrt{2 \cdot m_{eff}} v_n < -f_c \end{cases}$$

Formula between particle i and particle j if $(0 < \delta_n < \delta_{cut})$:

$$\mathbf{f}_{ij} = \left(\frac{f_c}{\delta_{cut}} \cdot \delta_n - f_c \right) \cdot \mathbf{n}$$

with $\mathbf{n}$ normalized vector from particle i to particle j

4.4.1.3 dmt adhesive normal law

DMT (Derjaguin–Muller–Toporov) is usually used to compute pull-off forces (force needed to split two rigid objects in contact). It is computed with global energy calculation. To define it, it needs an additional parameter representing an energy :

Constant	Description
γ	Adhesion energy per unit of surface

In case of contact, an additional force f_{DMT} is added to the normal force such that :

$$f_n = f_n - f_{DMT}$$

with:

$$f_{DMT} = 2\pi R_{eff} \gamma$$

where the effective radius R_{eff} is obtained from the respective radii R_i and R_j of the particles i and j:

$$R_{eff} = \frac{R_i \cdot R_j}{R_i + R_j}$$

and γ is the surface energy (linked to Van der Waals forces).

Note

Since DMT force law is added to the normal force, it requires the definition of a pure contact law, such as hooke one.

4.4.2 Contact Law Operators

Contact Law Operators define the type of contact law and the parameters associated with.

The operator naming convention follows the rule below:

```
contact_[material_mode]_[particle_type]_[pure_contact_law]_[cohesion_law]
```

Where each field is defined as follows:

Table 4: Operator name fields

Field	Possible values	Description
[material_mode]	singlemat, multimat	Material interaction mode
[particle_type]	sphere, polyhedron	Type of particles
[pure_contact_law]	hooke	Pure contact law
[cohesion_law]	cohesive, dmt, none	Cohesion / adhesion law

Note

For convenience, simplified operator names are available. They are strict equivalents of the full naming convention.

Sphere operators

- `contact_sphere` is equivalent to `contact_singlemat_sphere_hooke` or `contact_singlemat_sphere_hooke_none`
- `contact_multimat_sphere` is equivalent to `contact_multimat_sphere_hooke_none`

Polyhedron operators

- `contact_polyhedron` is equivalent to `contact_singlemat_polyhedron_hooke` or `contact_singlemat_polyhedron_hooke_none`
- `contact_multimat_polyhedron` is equivalent to `contact_multimat_polyhedron_hooke_none`

In a contact operator, the following parameters can be defined:

<i>symetric</i>	Activate or disable symmetric updates (do not disable it with polyhedron).
<i>config</i>	Data structure that contains contact force parameters (dncut, kn, kt, kr, fc, mu, damp_rate). Type = <code>exaDEM::ContactParams</code> . No default parameter.
<i>config_driver</i>	Data structure that contains contact force parameters (dncut, kn, kt, kr, fc, mu, damp_rate). Type = <code>exaDEM::ContactParams</code> . This parameter is optional.
<i>save_interac</i>	Store interactions into the classifier data structure. Default is false.

Here are 4 examples with YAML:

```
- contact_sphere:
  symetric: true
  config: { kn: 100000, kt: 100000, kr: 0.1, mu: 0.9, damp_rate: 0.9}
```

```
- contact_polyhedron:
  config: { kn: 10000, kt: 10000, kr: 0.1, mu: 0.1, damp_rate: 0.9}
  config_driver: { kn: 10000, kt: 10000, kr: 0.1, mu: 0.3, damp_rate: 0.9}
```

```
- contact_singlemat_sphere_hooke_cohesive:
  symetric: true
  config: { dncut: 0.1 m, kn: 100000, kt: 100000, kr: 0.1, fc: 0.05, mu: 0.9, damp_
↪ rate: 0.9}
```



```
- contact_singlemat_polyhedron_hooke_cohesive:
  config: { dncut: 0.1 m, kn: 10000, kt: 10000, kr: 0.1, fc: 0.05, mu: 0.1, damp_rate: 0.9 }
  config_driver: { dncut: 0.1 m, kn: 10000, kt: 10000, kr: 0.1, fc: 0.05, mu: 0.3, damp_rate: 0.9 }
```

```
- contact_singlemat_sphere_hooke_dmt:
  config: { kn: 1000, kt: 1000, kr: 0.1, mu: 0.2, damp_rate: 0.9, gamma: 10. }
  config_driver: { kn: 1000, kt: 800, kr: 0.1, mu: 0.5, damp_rate: 0.9, gamma: 10. }
```

Note

It is important to check that interaction lists have been built with this option enabled. By default, *exaDEM* always builds interaction lists using the symmetry option to limit the number of calculations.

Note

Contact Force With Cohesion ([cohesion_law] = *cohesive*) operator includes a cohesion force from *rcut* to *rcut+dncut* with the cohesion force parameter *fc*.

Note

- The `contact_*_sphere_*_*` operators are designed to process interactions built in `nbh_sphere` (please, include the `config_spheres.msp` file).
- The `contact_*_polyhedron_*_*` operators are designed to process interactions built in `nbh_polyhedron` (please, include the `config_polyhedra.msp` file).

4.4.3 Multi-Material

In the previous section, the contact law used the same parameters for all interactions ([material_mode] = *singlemat*). It is also possible to specify the contact law depending on the type of interaction ([material_mode] = *multimat*). In this section, we introduce how to define the values of the contact law between particles, as well as between particles and drivers.

Note

Unlike in some other DEM codes, the coefficients are **not** derived from the material properties (such as Poisson's ratio and Young's modulus).

Note

Since *exaDEM*-1.2.3, multi-material contact parameters are indexed by *group* (an integer, see the *group* particle field in the Particle Fields page) instead of directly by *particle type*. Before 1.2.3, the *particle type* was used directly. This decouples the shape/material identity (*type*) from the contact-law identity (*group*): several particle types can share the same group and therefore reuse the same contact parameters. Groups must be assigned beforehand, either with the *group* parameter of `set_fields` or with the dedicated `set_group` operator (see the Particle

Fields page).

To handle multiple particle types, you must use either the `contact_multimat_[]_[]_[]` operators as illustrated below.

- **YAML example for polyhedra:**

```
compute_force:
- gravity_force
- contact_multimat_polyhedron
```

- **YAML example for spheres:**

```
compute_force:
- gravity_force
- contact_multimat_sphere:
  symetric: true
```

The following examples illustrate the definition of contact parameters for two particle groups (**group 0**, **group 1**) and a driver identified by **0**.

4.4.3.1 Particle-Particle

- **Operator Name:** `multimat_contact_params`
- **Description:** This operator defines the contact law parameters between different particle groups.

Pa-ram-eter	Description
group1	List of the first particle group indice(s).
group2	List of the second particle group indice(s).
kn	Normal force coefficient for the specified interaction type.
kt	Tangential force coefficient for the specified interaction type.
kr	Rolling resistance coefficient for the specified interaction type.
mu	Friction coefficient for the specified interaction type.
dampre	Damping rate coefficient for the specified interaction type.
default	Applies the same parameter set to all undefined interaction configurations.
n_group	Number of distinct groups (INPUT_OUTPUT, since 1.2.3). Usually left unset: it is then deduced from group1/group2 (max index + 1). If it was already set upstream (e.g. by <code>set_group</code>), it is checked against group1/group2 instead, and a <code>default_config</code> must be provided to cover any group pair not listed explicitly.

Note

group1/group2 replace the former mat1/mat2 parameters (pre-1.2.3, which took particle type names directly). Particle groups must be assigned beforehand via the `group` parameter of `set_fields` or the `set_group` operator.

YAML example:

```
- multimat_contact_params:
  group1: [ 0, 0, 1 ]
  group2: [ 0, 1, 1 ]
  kn: [ 5000, 10000, 15000 ]
  kt: [ 4000, 8000, 12000 ]
  kr: [ 0.0, 0.0, 0.0 ]
  mu: [ 0.1, 0.2, 0.3 ]
  damprate: [ 0.999, 0.999, 0.999 ]
```

With *default_config*:

```
- multimat_contact_params:
  group1: [ 0 ]
  group2: [ 0 ]
  kn: [ 5000 ]
  kt: [ 4000 ]
  kr: [ 0.0 ]
  mu: [ 0.1 ]
  damprate: [ 0.999 ]
  default_config: { kn: 15000, kt: 12000, kr: 0.0, mu: 0.3, damp_rate: 0.999 }
```

A complete example is available (please report if the link does not work): [rotating-multimat.msp](#)

4.4.3.2 Particle-Driver

- **Operator Name:** `drivers_contact_params`
- **Description:** This operator defines the contact law parameters between particles and drivers.
- **Parameters:**

Parameter	Description
group	List of particle group indice(s) concerned by the interaction.
driver_id	Identifier(s) of the driver(s) interacting with the particles.
kn	Normal force coefficient for the specified interaction type.
kt	Tangential force coefficient for the specified interaction type.
kr	Rolling resistance coefficient for the specified interaction type.
mu	Friction coefficient for the specified interaction type.
damprate	Damping rate coefficient for the specified interaction type.
default_config	Applies the same parameter set to all undefined interaction configurations.

Note

`group` replaces the former `mat` parameter (pre-1.2.3, which took particle type names directly). Particle groups must be assigned beforehand via the `group` parameter of `set_fields` or the `set_group` operator.

```
- drivers_contact_params:
  group: [ 0, 1 ]
  driver_id: [ 0, 0 ]
  kn: [ 10000, 15000 ]
  kt: [ 8000, 12000 ]
```

(continues on next page)

(continued from previous page)

```
kr:      [ 0.0, 0.1 ]
mu:      [ 0.5, 0.5 ]
damprate: [ 0.999, 0.999 ]
```

A complete example is available (please report if the link does not work): [rotating-multimat.msp](#)

4.4.3.3 Checking Group Completeness

- **Operator Name:** `check_group_completeness` (since exaDEM-1.2.3)
- **Description:** Verifies that the number of groups configured in `multimat_cp` (via `multimat_contact_params` and/or `drivers_contact_params`) is large enough to cover the maximum `group` index actually present on particles. The simulation stops if a particle references a group for which no contact parameters were set up.
- **Parameters:** none.

```
- check_group_completeness
```

Note

This operator has no effect (it silently returns) if `multimat_cp` has not been produced yet, so it can safely be placed anywhere after `multimat_contact_params` and/or `drivers_contact_params` in the operator chain, typically right before `compute_force`.

4.4.4 External Forces

External forces are additional influences acting on particles within the simulation environment, originating from sources outside the particle system itself. These forces can include environmental factors like wind, fluid flow, or magnetic fields, as well as user-defined forces applied to specific particles or regions.

4.4.4.1 Gravity Operator

Formula:

$$\mathbf{f} = m \cdot \mathbf{g} \quad (4.1)$$

With $\mathbf{f}$ the forces, m the particle mass, and $\mathbf{g}$ the gravity constant.

- **Operator Name:** `gravity_force`
- **Description:** This operator computes forces related to the gravity.
- **Parameter:**

<i>gravity</i>	Define the gravity constant in function of the gravity axis, default value are x axis = 0, y axis = 0 and z axis = -9.807
----------------	---

YAML example:

```
- gravity_force:
  gravity: [0, 0, -0.009807]
```

4.4.4.2 Quadratic Drag Force

Formula:

$$\mathbf{f} = -\mu \cdot cx \cdot \|v\| \cdot \mathbf{v}$$

With $\mathbf{f}$ the particle forces, cx the aerodynamic coefficient, and μ the drag coefficient, $\|v\|$ the norm of the particle velocity, and $\mathbf{v}$ the particle velocity.

- Operator Name: `quadratic_force`
- Description: External forces that model air or fluid, $f = -\mu * cx * \text{norm}(v) * \text{vector}(v)$.
- Parameter:

<code>cx</code>	aerodynamic coefficient, default value is for air = 0.38.
<code>mu</code>	drag coefficient. default value is for air = 0.000015.

YAML example: see example *quadratic-force-test/QuadraticForceInput.msp*

```
- quadratic_force:
  cx: 0.38
  mu: 0.0000015
```

4.4.4.3 Fluid Grid Force

- Operator Name: `sphere_fluid_friction`
- Description: External forces that model a fluid computed from a grid such as: $f = \|f_v - p_v\|$

$$dv = f_v - p_v$$

$$f = cx \cdot dv \cdot \|dv\| \cdot \pi \cdot r \cdot r$$

With f_v the fluid velocity, p_v the particle velocity, r the particle radius, and cx a coefficient set to 1 by default.

Note

The fluid velocity f_v for each point of the grid has been defined by the operator `set_cell_values` (pure exaNBody operator).

4.4.5 Inner Bond Forces

4.4.5.1 Description

The `inner_bond_force` law models a **cohesive bond** between two contacting polyhedron faces, combining a linear elastic + viscous **normal** force, a linear elastic **tangential** force accumulated over the contact history, and a **fracture criterion** that breaks the bond once the stored elastic energy exceeds a material-dependent threshold.

With overlap $\delta = d_n - d_{n0}$ and relative normal velocity v_n :

$$f_n = -k_n w \delta + \gamma v_n$$

$$f_t = w k_t \mathbf{t}_{ds}$$

where w is the per-interaction weight and γ is the viscous damping coefficient derived from `damp_rate`. The stored elastic energies (tension only) are:

$$E_n = \frac{1}{2} w k_n \delta^2$$

$$E_t = \frac{1}{2} w k_t \|\mathbf{t}_{ds}\|^2$$

4.4.5.2 Parameters

Name	Unit	Description
kn	force/leng	Normal stiffness coefficient.
kt	force/leng	Tangential stiffness coefficient.
damp_rate	dimensionless	Normal damping ratio.
g	energy/area	MixedMode fracture energy release rate. Mutually exclusive with gn/gt. Breaks when $E_n + E_t > 2 A g$.
gn	energy/area	SeparateModes normal fracture energy release rate (requires gt). Breaks when $E_n > 2 A g_n$.
gt	energy/area	SeparateModes tangential fracture energy release rate (requires gn). Breaks when $E_t > 2 A g_t$.

Note

Exactly one fracture mode must be configured: g alone (MixedMode), or gn together with gt (SeparateModes). Mixing the two, or omitting both, is rejected at parsing time.

4.4.5.3 Usage

Single material, mixed-mode fracture:

```
inner_bond_params:
```

```
kn: 10000 N/m
kt: 8000 N/m
damp_rate: 0.1
g: 5.0 J/m^2
```

Single material, separate-mode fracture:

```
inner_bond_params:
```

```
kn: 10000 N/m
kt: 8000 N/m
damp_rate: 0.1
gn: 5.0 J/m^2
gt: 2.0 J/m^2
```

Multi-material simulations use the `inner_bond_params` operator with one entry per (group1[p], group2[p]) pair in parallel arrays:

Note

Since exaDEM-1.2.3, pairs are defined with group1/group2 (particle group indices) instead of the former mat1/mat2 (particle type names). Particle groups must be assigned beforehand via the group parameter of `set_fields` or the `set_group` operator.

```
- inner_bond_params:
  group1: [ 0, 0, 1 ]
  group2: [ 0, 1, 1 ]
```

(continues on next page)

(continued from previous page)

```

kn:      [ 5000, 10000, 15000 ]
kt:      [ 4000,  8000, 12000 ]
damp_rate: [ 0.999, 0.999, 0.999 ]
gn:      [ 1e-5, 1e-5, 1e-5 ]
gt:      [ 1e-5, 1e-5, 1e-5 ]

# or, mixed mode for every pair:
- inner_bond_params:
  group1: [ 0, 1 ]
  group2: [ 0, 1 ]
  kn:     [ 5000, 15000 ]
  kt:     [ 4000, 12000 ]
  damp_rate: [ 0.999, 0.999 ]
  g:      [ 1e-5, 1e-5 ]

```

4.4.5.4 See also

- `stick_polyhedra` — builds the inner-bond interactions and computes the fracture criterion's area term per bonded face.
- `apply_interface_fracture_criterion` — applies the configured rupture mode (mixed or separate) to decide whether a bonded interface breaks.

4.5 Numerical Scheme

4.5.1 Velocity Verlet Scheme

The time integration in `exaDEM` is performed using the velocity form of the Störmer-Verlet time integration algorithm, well-known as the *velocity-Verlet* algorithm. It is advantageous for DEM simulations due to its simplicity, stability, and accuracy. It provides a straightforward method for integrating Newton's equations of motion, efficiently calculating both positions and velocities. The *velocity-verlet* algorithm is integrated using the following scheme at each time step:

1. Calculate the position vector at full timestep:

$$\mathbf{x}(t + \Delta t) = \mathbf{x}(t) + \mathbf{v}(t) \Delta t + \mathbf{a}(t) \frac{\Delta t^2}{2}$$

2. Calculate the velocity vector at half timestep:

$$\mathbf{v}\left(t + \frac{\Delta t}{2}\right) = \mathbf{v}(t) + \mathbf{a}(t) \frac{\Delta t}{2}$$

3. Compute the acceleration vector at full time-step $\mathbf{a}\left(t + \Delta t\right)$ from the interatomic potential using the position at full timestep $\mathbf{x}\left(t + \Delta t\right)$
4. Finally, calculate the velocity vector at full timestep:

$$\mathbf{v}(t + \Delta t) = \mathbf{v}\left(t + \frac{\Delta t}{2}\right) + \mathbf{a}(t + \Delta t) \Delta t$$

In `exaDEM`, the numerical scheme definition can be found in `exaDEM/data/config/config_numerical_schemes.msp` and the YAML block for the Velocity-Verlet scheme reads

```

numerical_scheme: numerical_scheme_verlet

numerical_scheme_verlet:
  name: scheme

```

(continues on next page)

(continued from previous page)

```
body:
- push_f_v_r: { dt_scale: 1.0 }
- push_f_v: { dt_scale: 0.5 }
- push_to_quaternion: { dt_scale: 1.0 }
- check_and_update_particles
- reset_force_moment
- compute_force_op
- force_to_accel
- update_angular_acceleration
- update_angular_velocity: { dt_scale: 1 }
- push_f_v: { dt_scale: 0.5 }
```

Note

Some operators are combined in operators to optimize code and avoid loading the same fields several times: `combined_compute_prolog` and `combined_compute_epilog`.

The `exaNBody` code provides a generic operator for 1st order time-integration purposes. For example, the file `exaNBody/src/exanb/push_vec3_1st_order_xform.cpp` provides 3 different variants:

- `push_v_r` : for updating positions from velocities
- `push_f_v` : for updating velocities from forces (i.e. acceleration)
- `push_f_r` : for updating positions from forces (i.e. acceleration)

In addition, the `exaNBody` code also provides a generic operator for 2nd order time-integration purposes. For example, the file `exaNBody/src/exanb/push_vec3_2nd_order_xform.cpp` provides the following variant:

- `push_f_v_r` : for updating positions from both velocities and forces (i.e. accelerations)

Since in `exaDEM` positions are expressed in a reduced frame, the argument `xform_mode: INV_XFORM` is mandatory when using any operator that updates the particles positions. The operators are described in detail in the following section.

4.5.2 Operators

In this section, we'll explain operators and how to use them. The idea is to be able to easily reorder / add / remove operators to easily build another time integration scheme than the Verlet Vitesse scheme. For performance reasons, some operators are merged into a single operator, such as `combined_compute_epilog` and `combined_compute_prolog`.

4.5.2.1 Reset Forces and Moments

- Operator Name: `reset_force_moment`
- Description: This operator resets two grid fields: moments and forces.

Here is a YAML example:

```
- reset_force_moment
```


4.5.2.2 Update Particle Acceleration

- Operator Name: `force_to_accel`
- Description: This operator computes particle accelerations from forces and mass.

Formula:

$$\mathbf{f} = \mathbf{a}.m;$$

with $\mathbf{f}$ the sum of forces applied to the particle, $\mathbf{a}$ the acceleration of the particle, and m its mass.

Here is a YAML example:

```
- force_to_accel
```

4.5.2.3 Update Particle Orientation

- Operator Name: `push_to_quaternion`
- Description: This operator computes particle orientations from angular velocities and angular accelerations.
- Parameter:
 - `dt_scale`: Coefficient applied to the increment time (Δ_t)

Formula:

$$Q = Q + Q.av.\Delta_t$$

$$Q = \frac{Q}{||Q||}$$

$$av = av + aa.\frac{\Delta_t^2}{2}$$

with aa the angular acceleration, av the angular velocity, and Q the particle orientation.

Here is a YAML example:

```
- push_to_quaternion: { dt_scale: 1.0 }
```

4.5.2.4 Update Angular Velocity

- Operator Name: `push_to_angular_velocity`
- Description: This operator computes particle angular velocity values from angular velocities and angular accelerations.
- Parameter:
 - `dt_scale`: Coefficient applied to the increment time (Δ_t)

Formula:

$$av = av + aa.\frac{\Delta_t^2}{2}$$

with aa the angular acceleration, av the angular velocity, and Q the particle orientation.

Here is a YAML example:

```
- push_to_angular_velocity: { dt_scale: 1.0 }
```

Note

This operator is not (directly) used, it has been merged in the operator `combined_compute_epilog`

4.5.2.5 Update Angular Acceleration

- Operator Name: `push_to_angular_acceleration`
- Description: This operator computes angular accelerations.

Formula:

$$\omega = \bar{Q}.av$$

$$aa = Q.\dot{\omega}$$

with **aa** the angular acceleration, **av** the angular velocity, **I** the particle inertia, and **Q** the particle orientation (and $\bar{Q}$ its conjugate). To compute $\dot{\omega}$, we need the particle moment and the particle inertia values.

Here is a YAML example:

```
- push_to_angular_acceleration
```

Note

This operator is not (directly) used, it has been merged in the operator `combined_compute_epilog`

4.5.2.6 Combined Prolog

- Operator Name: `combined_compute_prolog`
- Description: This is an operator that combines 3 operators:
 - `push_f_v_r`
 - `push_f_v`
 - `push_to_quaternion`
- Parameter:
 - `dt_scale`: Coefficient applied to the increment time (Δ_t)

Here is a YAML example:

```
- combined_compute_prolog
```

4.5.2.7 Combined Epilog

- Operator Name: `combined_compute_epilog`
- Description: This is an operator that combines 3 operators:
 - `push_to_angular_acceleration`
 - `push_angular_velocity`

– push_f_v

Here is a YAML example:

```
- combined_compute_epilog
```

4.6 Drivers

This section provides an overview of the **drivers** implemented in ExaDEM, followed by a detailed description of each **driver** integrated into the system. In addition, this section presents the **driver-extractor** tool (see *Driver Extractor*), which makes it possible to output field values.

4.6.1 Quick Overview of Drivers

4.6.1.1 What is a driver in ExaDEM

In ExaDEM, a **driver** denotes a significant shape, such as walls, cylinders, or other geometric forms, which are treated differently from particles due to their unique characteristics. While the ExaDEM framework has been optimized and parallelized to effectively handle particles and polyhedra of comparable scales, **drivers** necessitate special treatment owing to their distinct properties. To accommodate multiple **drivers** in a simulation, ExaDEM offers flexibility through the creation of a **drivers** list. This feature allows users to include any number of **drivers** required to define boundary conditions and obstacles within the simulation environment.

The current implementation of ExaDEM includes a variety of **drivers**, each serving specific purposes within the simulation framework. The detailed list of implemented **drivers** is provided in the following table:

Table 5: Glossary Of Drivers

Name	Driver Type	Operator Name
Cylinder	CYLINDER	register_cylinder
Wall / Surface	SURFACE	register_surface
Ball / Sphere	BALL	register_ball
RShape	RShapeDriver	register_stl_mesh/ register_rshape
Undefined	UNDEFINED	no operator

Note

When adding a **driver** to the simulation in ExaDEM, it is essential to define a contact parameter list specific to the **driver** within the *compute_contact_interaction* operator.

4.6.1.2 Common Driver Parameters

Drivers share common parameters contained in the `Driver_params` class. These parameters are mainly used in the time integration scheme to detect the type of displacement. In the tables below, we describe the types of motion available by drivers as well as a list of motions available on the driver:

Table 6: Glossary Of Motion Types

Motion type	Description
STATIONARY	Stationary state with no translational motion, allowing rotations of Drivers.
LINEAR_MOTION	Linear movement type, straight-line motion at constant velocity
COMPRESSIVE_FORCE	Movement influenced by compressive forces, depending on driver type.
LINEAR_FORCE_MOTION	Linear motion driven by constant acceleration, incorporating the effects of the sample's resultant force.
FORCE	General movement caused by applied forces. Act likes a particle (R-shape).
LINEAR_COMPRESSIVE_MOTION	Linear movement combined with compressive forces.
TABULATED	Motion defined by precomputed or tabulated data.
SHAKER	Oscillatory or vibratory motion, typically mimicking a shaking mechanism.
PENDULUM_MOTION	Oscillatory swinging around a suspension point (pendulum-like).
EXPRESSION	Impose a velocity or angular velocity field defined by a user-specified analytical expression. This option use the library tinyexpr: Github

Table 7: Glossary Of Motion Types per Driver

Motion Type	STATIC	LINEAR	COMPRESSIVE	LINEAR FORCE	LINEAR TABULATED	SHAKER	PENDULUM	EXPRESSION
Cylinder	✓					✓		
Surface	✓	✓			✓	✓	✓	
Ball	✓	✓	✓			✓		
Std Mesh	✓	✓		✓	✓	✓		✓

For all these types of movement, the drivers adopt a velocity Verlet integration time scheme. Below is a summary table showing how positions, forces, or velocities are calculated according to the type of movement.

STATIONARY

$$P = P$$

$$V = 0$$

with P the driver position, V the driver velocity.

LINEAR_MOTION

$$V = M_{vector} \cdot C_{velocity}$$

with V the driver velocity, M_{vector} the motion vector, and $C_{velocity}$ the value of the [constant] velocity.

COMPRESSIVE_FORCE

For spheres (or Ball), we adapt the scalar radius value (radius (R), $R_{velocity}$, $R_{acceleration}$):

$$R_{acceleration} = \frac{F_{driver} - \sigma \cdot S - \text{damprate} \cdot R_{velocity}}{0.5 \cdot m_{system}}$$

with S , the driver surface, m_{system} the mass of the system, and $math:damprate$ the damprate (TODO complete).

LINEAR_FORCE_MOTION

$$F = (\text{dot}(F_{driver}, M_{vector}) + C_F) \cdot M_{vector}$$

with F the driver forces, F_{driver} the sum of the forces applied to the driver by the particles, C_F the value of the [constant] force, and M_{vector} the motion vector.

FORCE

$$F = F_{driver} + gravity$$

with F the driver forces and F_{driver} the sum of the forces applied to the driver by the particles.

LINEAR_COMPRESSIVE_MOTION

$$F = (\text{dot}(M_{vector}, F_{driver}) - \sigma \cdot S - \text{damprate} \cdot V) \cdot M_{vector}$$

with F the driver forces, V the driver velocity, S , the driver surface, $damprate$ the damprate (TODO complete), and M_{vector} the motion vector.

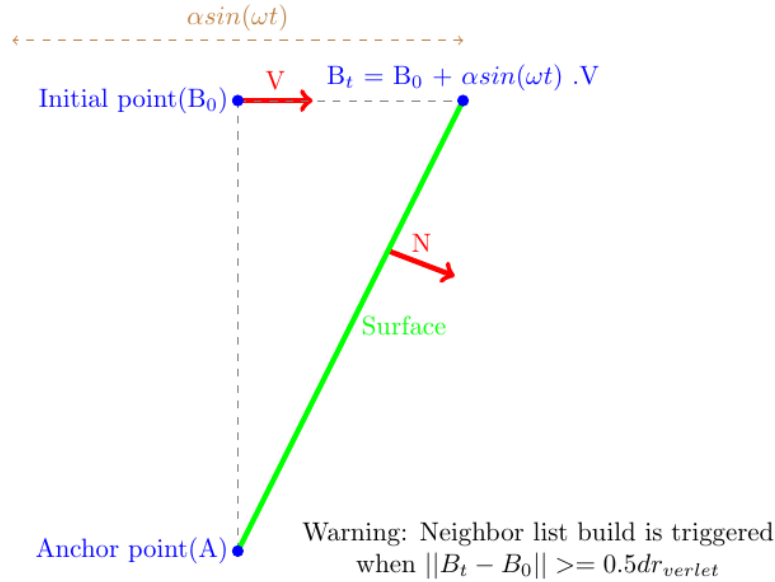
SHAKER

$$C = A \cdot \sin(\omega T) \cdot N_{shaker} + C_{initial},$$

$$V = \omega \cdot A \cdot \cos(\omega T) \cdot N_{shaker},$$

with C the driver center, $C_{initial}$ the initial driver center at $T = t - \text{motion_start_threshold}$, $math:N_{shaker}$ the shaker normal vector, $math:V$ the driver velocity, A the signal amplitude, and ω , the signal frequency.

PENDULUM_MOTION



$$B_t = A \cdot \sin(\omega T) \cdot D + B_0, V = 0$$

with B_0 the initial position, B_t the current position at $T = t - \text{motion_start_threshold}$, $math:D$ the swing normal vector, $math:V$ the driver velocity, A the signal amplitude, ω , the signal frequency, A , the anchor point (A is invariant and always intersect the surface), N the normal of the surface (if it's a surface) or the new direction of the object.

EXPRESSION

$$V = Formula(t)$$
$$V_{rot} = Formula(t)$$

with `math:V` the driver velocity and `math:V_rot` the angular driver velocity. Formula are defined in the struct `Driver_expr` (`expr`) in the slot `params`.

And keywords:

- `motion_vector`: M_{vector} , slot params
- `const_vel`: $C_{velocity}$, slot params
- `const_f`: C_F , slot params
- `damprate`: $damprate$, slot params
- `amplitude`: A , slot params
- `omega`: ω , slot params
- `shaker_dir`: N_{shaker} , slot params
- `pendulum_anchor_point`: $Anchor$, slot params
- `pendulum_initial_position`: B_0 , slot params
- `pendulum_swing_dir`: D , slot params

4.6.1.3 Add a Driver To Your Simulation

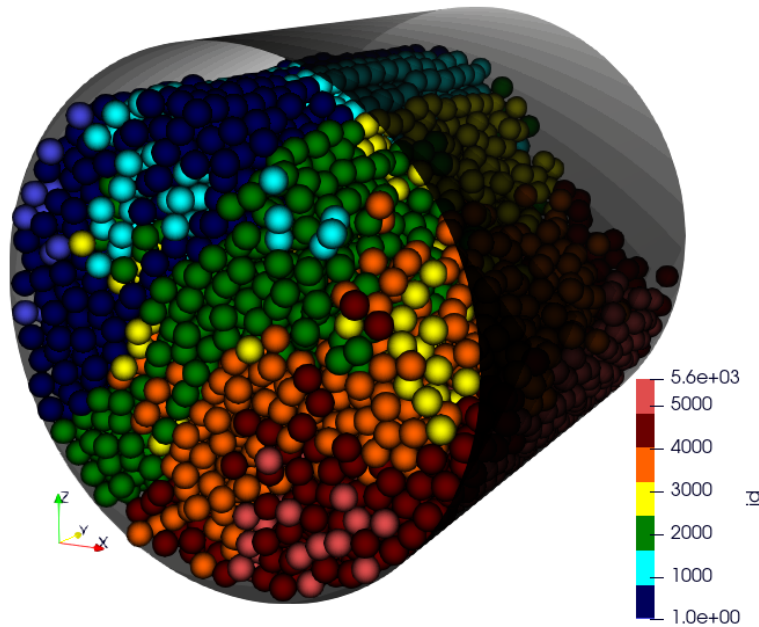
In ExaDEM, drivers are managed differently depending on whether spheres or polyhedra are used in the simulation. Forces are computed per interaction for polyhedra, while forces are computed and summed per sphere body:

- Using spheres, a special contact force is added to handle interactions with drivers in the `contact_force_driver` operator.
- Using polyhedra, special interactions (described in the Polyhedra section) are added to the interaction lists. Additionally, you need to specify your driver list in the list of operators called `setup_drivers`, which is integrated into the default ExaDEM execution. It's crucial to specify an ID for each driver. If you create a second driver with an already used ID, it will overwrite the previous driver configuration.

In the following sections, we provide brief descriptions of available drivers. Please note that a test case is defined for each driver in the `example` directory.

4.6.2 Rotating Drum / Cylinder

The rotating drum or cylinder driver represents an infinite cylinder rotating along a specified axis. It is defined by parameters including its middle, velocity, axis, and angular velocity.



- Operator name: `register_cylinder`
- Description: This operator adds a cylinder to the drivers list.
- Parameters:
 - *id*: Driver index
 - *state*: Current cylinder state, default is {radius: REQUIRED, axis: REQUIRED, center: REQUIRED, vel: [0,0,0], vrot: [0,0,0], rv: 0, ra: 0}. You need to specify the radius and center.
 - *params*: List of params, motion type, motion vectors Default is { motion_type: STATIONARY }.

YAML example:

```
- register_cylinder:
  id: 0
  state: {center: [2.5, 2.5, 2.5], axis: [1, 0, 1], radius: 4}
  params: { motion_type: STATIONARY }
```

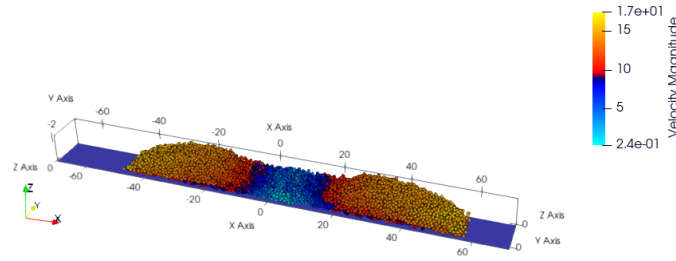
4.6.2.1 SHAKER mode

```
- register_cylinder:
  id: 0
  state: {center: [25, 25, 25], axis: [1, 0, 1], radius: 25}
  params: { motion_type: SHAKER, amplitude: 0.5, omega: 1., shaker_dir: [0, 0, 1] }
```

This example is available at: `exaDEM/example/spheres/shaker_cylinder/shaker_cylinder.msp`

4.6.3 Wall / Surface

The wall or surface driver represents an infinite wall within the simulation environment. It is defined by parameters including its normal vector, offset, and velocity. Please note that currently, no angular velocity is defined for this driver.



- Operator name: `register_surface`
- Description: This operator adds a surface/wall to the drivers list.
- Parameters:
 - *id*: Driver index
 - *state*:
 - *params*:

4.6.3.1 STATIONARY mode

```
- register_surface:
  id: 0
  state: {normal: [0,0,1], offset: -0.5}
  params: { motion_type: STATIONARY }
```

4.6.3.2 LINEAR_MOTION mode

```
- register_surface:
  id: 4
  state: { normal: [1,0,0], offset: 11}
  params: { motion_type: LINEAR_MOTION, motion_vector: [-1,0,0], const_vel: 2.5 }
```

This example is available at: `exaDEM/example/spheres/rigid-surface/rigid_surface_linear_motion.msp`

Note

motion vector can only be equal to normal and -normal.

4.6.3.3 LINEAR_COMPRESSIVE_MOTION mode

```
- register_surface:
  id: 4
  state: { normal: [1,0,0], offset: 11, surface: 144}
  params: { motion_type: LINEAR_COMPRESSIVE_MOTION, motion_vector: [1,0,0], sigma: 0.5,
↳damprate: 0.999 }
```

This example is available at: `exaDEM/example/spheres/rigid-surface/compression_wall.msp`

This example is available at: `exaDEM/example/polyhedra/rigid_surface/rigid_surface_compression_motion.msp`

Note

- If you have chosen the “LINEAR_COMPRESSIVE_MOTION” mode, you will need to define the value of the wall surface.
- The `motion_vector` is set to normal.
- When `> 0` and the surface is free of external forces, the wall displaces in the direction opposite to the surface normal.

4.6.3.4 SHAKER mode

```
- register_surface:
  id: 2
  state: {normal: [0,0,1], offset: -1}
  params:
    motion_type: SHAKER
    amplitude: 0.5
    omega: 1e1
    motion_start_threshold: 3.0
    motion_end_threshold: 8.0
```

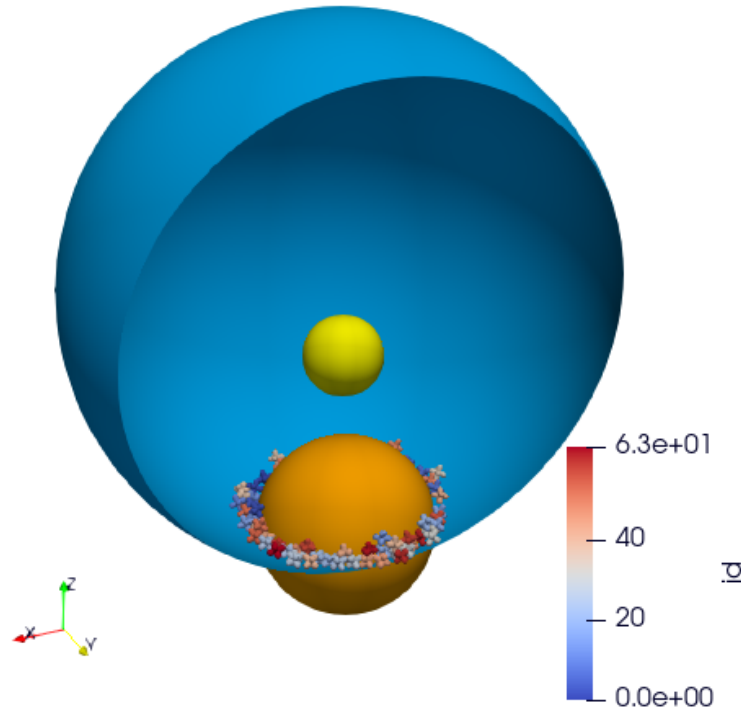
4.6.3.5 PENDULUM_MOTION mode

```
- register_surface:
  id: 3
  state: { normal: [1,0,0], offset: -1} # ignored
  params: { motion_type: PENDULUM_MOTION, amplitude: 3, omega: 4, pendulum_anchor_
↳point: [-1,5,-1], pendulum_initial_position: [-1, 5, 11], pendulum_swing_dir: [1, 0,
↳0] }
```

See example: `example/spheres/rigid-surface/rigid_surface_pendulum_motion.msp`

4.6.4 Ball / Sphere

The ball or sphere driver represents a spherical object within the simulation environment. It is defined by parameters including its center, velocity, and angular velocity. This driver can be utilized as a boundary condition or obstacle in the simulation.



- Operator name: `register_ball`
- Description: This operator adds a ball / sphere (boundary condition or obstacle) to the drivers list.
- Parameters:
 - *id*: Driver index
 - *state*: Current ball state, default is {radius: REQUIRED, center: REQUIRED, vel: [0,0,0], vrot: [0,0,0], rv: 0, ra: 0}. You need to specify the radius and center.
 - *params*: List of params, motion type, motion vectors Default is { motion_type: STATIONARY}.

Note

- *ra* is the “radius acceleration” and *rv* the “radius velocity” used during the radial compression, i.e. shrinking or stretching the radius of a ball until the desired pressure is reached between the ball and the particles inside. This requires the motion type COMPRESSIVE_FORCE.
- If the motion type is LINEAR_MOTION, the velocity (*vel*) is computed from *motion_vector* and *const_vel*.
- If the motion type is COMPRESSIVE_FORCE, the velocity (*vel*) is set to 0.

4.6.4.1 STATIONARY mode

YAML example:

```
- register_ball:
  id: 2
  state: {center: [2,2,-20], radius: 7}
  params: { motion_type: STATIONARY }
```

See: exaDEM/example/spheres/ball/driver-ball-stationary.msp

4.6.4.2 LINEAR MOTION mode

```
- register_ball:
  id: 1
  state: {center: [30,2,-10], radius: 8}
  params: { motion_type: LINEAR_MOTION , motion_vector: [-1,0,0], const_vel: 0.5}
```

See: exaDEM/example/spheres/ball/driver-ball-linear.msp

4.6.4.3 COMPRESSIVE_FORCE mode

```
- register_ball:
  id: 0
  state: {center: [0,0,0], radius: 11}
  params: {motion_type: COMPRESSIVE_FORCE , sigma: 1.0, damprate: 0.999}
```

See: exaDEM/example/spheres/ball/driver-ball-radial-stress.msp

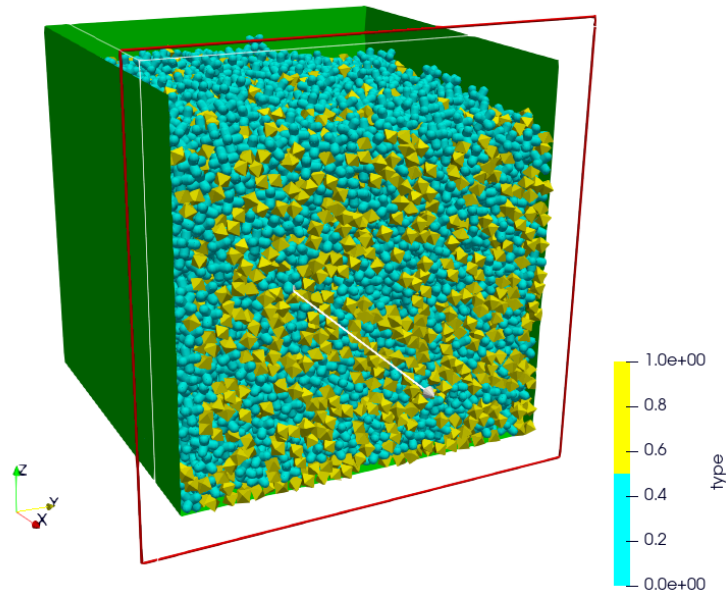
4.6.4.4 TABULATED mode

```
- register_ball:
  id: 0
  state: {center: [0,0,0], radius: 7}
  params:
    motion_type: TABULATED
    time: [0, 25, 50, 75]
    positions: [[-20,0,-20], [20,0,-20], [20, 0, -15], [-20, 0, -15]]
```

See: exaDEM/example/spheres/ball/driver-ball-tabulated.msp

4.6.5 RShape / Polyhedron

The RShape driver is constructed from a *.stl* (Stereolithography) or a *.shp* (Shape file) file to create a mesh of faces. This approach enables the rapid setup of complex geometries within the simulation environment. It's important to note that faces in a RShape driver are processed as a sphere polyhedron, meaning a small layer is added around each face.



- Operator name: `register_stl_mesh/register_rshape`
- Description: This operator adds a “RShape” to the drivers list.
- Parameters:
 - *id*: Driver index
 - *filename*: Input filename (.stl or .shp)
 - *minskowski*: Minskowski radius value
 - *binary*: Define if the stl file is ascii or binary, default is false. Ignored if the format file is a .shp.
 - *scale*: Define the scale factor of applied to the shape, default is 1.
 - *deform*: Define the non-uniform deformation factors along x, y, and z axes. Format is [x, y, z]. Values must be strictly positive. Default is [1, 1, 1].
 - *state*: Define the center, velocity, angular velocity and the orientatation. Default is: state: {center: [0,0,0], vel: [0,0,0], vrot: [0,0,0], quat: [1,0,0,0]}.
 - *params*: List of params, motion type, motion vectors Default is { motion_type: STATIONARY}.

YAML examples:

4.6.5.1 STATIONARY mode

```
- register_stl_mesh:
  id: 0
  binary: false
  filename: mesh.stl
  minskowski: 0.001 m
  params: {motion_type: STATIONARY}
```

4.6.5.2 LINEAR_MOTION mode

```
- register_rshape:
  id: 0
  state: {center: [0.4,0,0]}
  params: { motion_type: LINEAR_MOTION, motion_vector: [-1,0,0], const_vel: 0.5 }
  filename: mesh.shp
  minskowski: 0.001 m
```

4.6.5.3 TABULATED mode

```
- register_stl_mesh:
  id: 0
  state: {}
  params:
    motion_type: TABULATED
    time: [0, 1, 1.5, 2]
    positions: [[0.4, 0, 0], [-1, 0, 0], [0.4, 0, 0], [0.4, 0, 0]]
  filename: mesh.stl
  minskowski: 0.001 m
```

4.6.5.4 LINEAR_FORCE_MOTION mode

```
- register_stl_mesh:
  id: 1
  filename: piston_haut.stl
  scale: 0.5002
  minskowski: 0.001
  state: { center: [0.0, 0.0, 9.], vel: [0,0,-0.025], quat: [1,0,0,0], mass: 1}
  params: { motion_type: LINEAR_FORCE_MOTION, motion_vector: [0,0,-1], const_force: 100,
  → }
```

Note

You need to define the mass of your driver.

4.6.5.5 LINEAR_COMPRESSIVE_MOTION mode

```
- register_stl_mesh:
  id: 1
  filename: piston_haut.stl
  scale: 0.5002
  minskowski: 0.001
  state: { center: [0.0, 0.0, 9.], vel: [0,0,-0.025], quat: [1,0,0,0], mass: 1,
  → surface: 1.6146970415e+02}
  params: { motion_type: LINEAR_COMPRESSIVE_MOTION, motion_vector: [0,0,-1], sigma: 0.5,
  → damprate: 0.5 }
```

Note

You will need to define the mass and the surface of your driver. If you don't specify a surface, *exaDEM* will propose to you a value corresponding to the sum of the face surfaces composing the RShape driver.

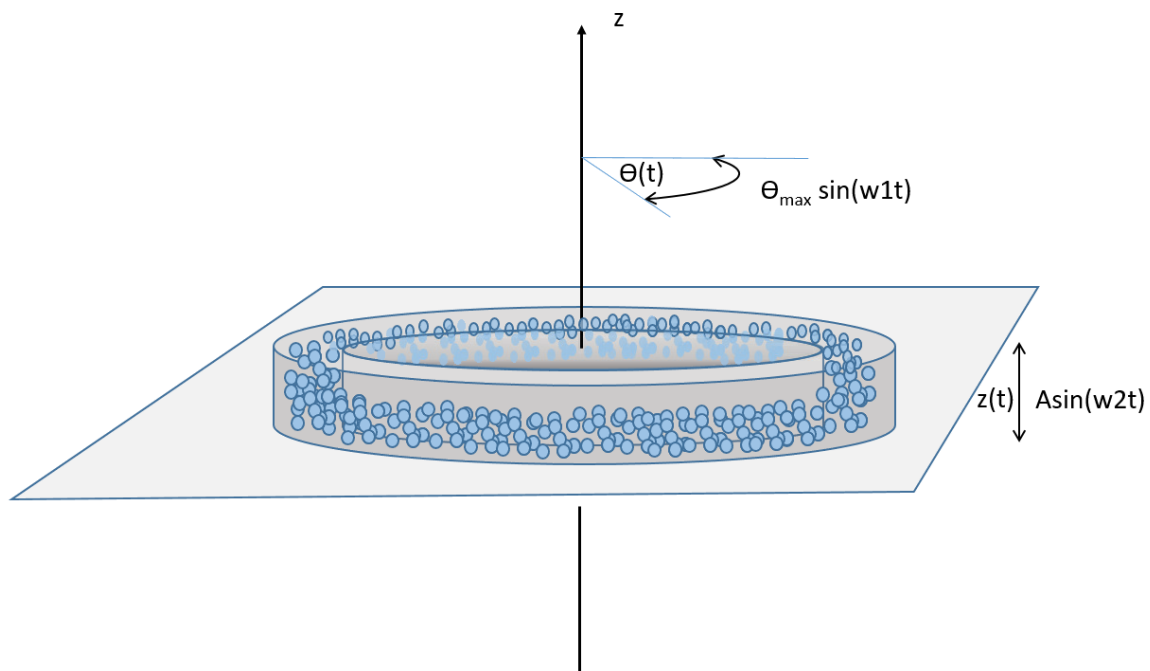
4.6.5.6 SHAKER mode

```
- register_rshape:
  id: 0
  minskowski: 0.1
  filename: shaker.stl
  binary: true
  state: {center: [0,0,0] }
  params: { motion_type: SHAKER, amplitude: 5, omega: 2 , shaker_dir: [0,1,0] , motion_
  start_threshold: 10 }
```

This example is available here: `exaDEM/example/spheres/shaker/shaker_stl.msp`

4.6.5.7 EXPRESSION mode

This example is available here (RSA plugins is required): `exaDEM/example/spheres/stl-mesh/tore_expression_motion.msp`



```
- register_stl_mesh:
  id: 0
```

(continues on next page)

(continued from previous page)

```

filename: tore.stl
minskowski: 0.01
state: { }
params:
  motion_type: EXPRESSION
  expr:
    vz: 2.5 * cos( 10 * t)
    vrotz: 0.5 * cos(20 * t)
binary: true

```

Example to impose moment:

```

- register_stl_mesh:
  id: 0
  filename: pale2.stl
  minskowski: 0.01
  binary: true
  state:
    vrot: [0,0,0]
    center: [2.5,2.5,2]
    quat: [0,-0.707107,-0.707107,0]
    mass: 49.0454
    inertia: [77.238167282, 388.401803042, 344.515979122]
  params:
    motion_type: EXPRESSION
    expr:
      momx: 0
      momy: 0
      momz: 5

```

This example is available at: [exaDEM/example/polyhedra/imposed_driver_moment/driver_expr_mom.msp](#). It corresponds to the example in the following section.

Another example is available at: [exaDEM/example/spheres/cylinder_stl](#) and is a funny variant of *Rotating drum simulation*.

4.6.5.8 Applying a Constant Moment with Sample Retroaction

It is possible to apply a constant moment while accounting for the retroaction exerted by the sample onto the driver. To enable this behavior, the driver definition must include the following physical properties:

- **center**: The rotation center of the driver.
- **inertia**: The rotational inertia of the driver.
- **mass**: The mass of the driver.

As with angular velocity control, this mechanism does **not** fall under the `motion_type` category.

Note

The applied moment is restricted to the direction defined in the `moment` slot. Only the component aligned with this direction is applied.

YAML example:

```
setup_drivers:
- register_stl_mesh:
  id: 0
  filename: pale2.stl
  minskowski: 0.01
  binary: true
  state: { vrot: [0,0,0], center: [2.5,2.5,2], quat: [0,-0.707107,-0.707107,0] },
  ↪ mass: 49.0454, inertia: [77.238167282, 388.401803042, 344.515979122], moment: [0,0,1] }
  ↪ # vrot: rad.s-1 normalement
  params: { motion_type: STATIONARY }
```

These examples are available at:

- [Right]: `example/polyhedra/imposed_driver_moment/driver_counter_mom.msp`
- [Left]: `example/polyhedra/imposed_driver_moment/driver_applied_mom.msp`

4.6.6 Modification of a Driver's Motion Type

This feature allows changing the motion type of a driver during a simulation. Each driver is identified by an *id*, assigned when the driver is registered. To enable motion changes, you can add one or more calls to `modify_motion` inside the `driver_motion_policy` YAML block (nop by default).

Operator name: `modify_motion`

Parameters:

- `id`: Index of the target driver.
- `motion`: Dictionary defining the new motion parameters (motion type, motion vectors, etc.). Example: `{ motion_type: STATIONARY }`.
- `time`: Simulation time at which the motion type should be changed.
- `display`: If true, prints details about the updated motion type (default: false).

YAML Example:

```
driver_motion_policy:
- modify_motion:
  id: 0
  time: 0.5
  motion: { motion_type: LINEAR_COMPRESSIVE_MOTION, motion_vector: [1,0,0], sigma: 5,
  ↪ damprate: 0.999 }
  display: true
- modify_motion:
  id: 0
  time: 1.5
  motion: { motion_type: STATIONARY }
```

(continues on next page)

(continued from previous page)

```

- modify_motion:
  id: 0
  time: 2.5
  motion: { motion_type: LINEAR_COMPRESSIVE_MOTION, motion_vector: [1,0,0], sigma: 5,
→ damprate: 0.999 }

- modify_motion:
  id: 0
  time: 4.0
  motion: { motion_type: LINEAR_MOTION, motion_vector: [1,0,0], const_vel: 3 }

```

Example available: `exaDEM/example/polyhedra/rigid_surface/rigid_surface_modify_motion.msp`

4.6.7 I/O Drivers

An input/output system has been implemented primarily for drivers performing movements, such as a rigid surface compressing a sample or a blade rotating around an axis.

The drivers' output is automatically triggered when the user sets the global variable: `simulation_dump_frequency`. This command also allows particles and interactions to be stored in a separate file. The drivers are then saved in a file located at `ExaDEMOutputDir/CheckpointFiles/driver_%010d.msp`, containing the drivers' information. In the case of an RShape driver, a `shp` file is added to the `ExaDEMOutputDir/CheckpointFiles/` directory, which contains the geometry of the RShape. To restart the driver along with your simulation, simply include the `.msp` file containing the `setup_driver` operator block at the beginning of your restart file.

YAML example:

```

grid_flavor: grid_flavor_dem
includes:
- config_spheres.msp
- ExaDEMOutputDir/CheckpointFiles/driver_0000040000.msp
global:
  simulation_dump_frequency: 10000

```

Similarly, ExaDEM saves RShape drivers each time a Paraview output is generated by setting the global variable: `simulation_paraview_frequency`. The RShape driver is then translated and oriented correctly in the `ExaDEMOutputDir/ParaviewOutputFiles/` directory as `shape_name_iteration.vtk`.

Another feature displays the driver summary. To do this, use the `print_drivers` operator, which is called by default when initializing an exaDEM simulation.

- Operator name: `print_drivers`
- Description: This operator prints drivers.

YAML example:

```

- print_drivers

```

Output example:

```

===== Driver Configurations =====
===== Summary

```

(continues on next page)

(continued from previous page)

```

Drivers Stats
Number of drivers: 3
Number of Cylinders: 1
Number of Surfaces: 0
Number of Balls: 0
Number of Stl_meshs: 2
Number of Undefined Drivers: 0
===== List Of Drivers
Driver [0]:
Driver Type: RSHAPE
Name   : base
Center : 0,0,-20
Vel    : 0,0,0
AngVel : 0,0,0
Quat   : 1 0 0 0
Number of faces   : 52
Number of edges   : 150
Number of vertices : 100
Driver [1]:
Driver Type: Cylinder
Radius: 25
Axis   : 1,1,0
Center: 0,0,0
Vel    : 0,0,0
AngVel: 0,0,0
Driver [2]:
Driver Type: RSHAPE
Name   : palefine
Center : 0,0,1.5
Vel    : 0,0,-0.0174
AngVel : 0,0,-0.004
Quat   : 1 0 0 0
Number of faces   : 25952
Number of edges   : 77856
Number of vertices : 31647
=====

```

4.6.8 Driver Extractor

The `driver_extractor` tool is used to retrieve specific quantities related to drivers in an output file. To do so, trackers are added and identified by the driver ID and the field that should be recorded.

The use of this tool is integrated into the default `exaDEM` workflow; however, if no tracker has been defined, it remains inactive and performs no operation.

4.6.8.1 Tracker Dictionary

The `driver_extractor` provides a set of predefined trackable fields that can be recorded for each driver during a simulation. These fields correspond to the main kinematic and dynamic quantities associated with the driver state. Each field is identified by a string key and associated with a specific data type.

The full list of available trackable fields is given below:

Table 8: Trackable driver fields

Name	Type	Description
type	int	Driver type identifier
rx	double	Position component along the x-axis
ry	double	Position component along the y-axis
rz	double	Position component along the z-axis
vx	double	Linear velocity component along the x-axis
vy	double	Linear velocity component along the y-axis
vz	double	Linear velocity component along the z-axis
vrotx	double	Angular velocity component around the x-axis
vroty	double	Angular velocity component around the y-axis
vrotz	double	Angular velocity component around the z-axis
fx	double	Force component along the x-axis
fy	double	Force component along the y-axis
fz	double	Force component along the z-axis
momx	double	Torque component around the x-axis
momy	double	Torque component around the y-axis
momz	double	Torque component around the z-axis

4.6.8.2 Adding a driver tracker

Trackers are used by the driver extractor to record specific fields associated with selected drivers during a simulation. Each tracker is defined by a driver identifier (*id*) and a list of fields to be extracted for that driver.

If no tracker is registered, the driver extractor remains inactive and produces no output.

Configuration is done through the `register_driver_extractor` operator as shown below:

```

setup_drivers:
...

input_data:
- register_driver_extractor:
  trackers:
    - id: 2
      fields: [vroty, fx, fy, fz, momy]
    - id: 0
      fields: [fy]
    - id: 1
      fields: [fy]
  verbosity: true

```

The *id* field corresponds to the unique identifier of the driver. The *fields* list specifies which quantities (see Trackable fields section) will be recorded for this driver.

When *verbosity* is enabled, the extractor prints additional information about the registered trackers during initialization, which can be useful for debugging and verification.

In addition, exaDEM displays the *driver extractor* after the initialization step:

```

===== Driver Extractor =====
Tracked Driver Id: 2 | Fields: [ vroty fx fy fz momy ]
Tracked Driver Id: 0 | Fields: [ fy ]

```

(continues on next page)

(continued from previous page)

Tracked Driver Id: 1 | Fields: [fy]

=====

4.6.8.3 Output format

The driver extractor writes its results to a single text file located at:

DriverExtractor/data.txt

This file contains a tabular time history of all requested tracker fields. Each column follows the naming convention:

field_driverId

where *field* is the tracked quantity name and *driverId* is the identifier of the corresponding driver.

The first column of the file is always the physical simulation time (*time*), which is included automatically regardless of the registered trackers.

An example of the generated output file is shown below:

```
time vroty_2 fx_2 fy_2 fz_2 momy_2 fy_0 fy_1
0.000500 0.250000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.005000 0.250000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.010000 0.250000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.015000 0.250000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.020000 0.250000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
...
9.980000 0.250000 -5.379081 -0.862643 -81.987323 -85.784585 19.196276 -17.884607
9.985000 0.250000 -8.693555 -0.686199 -81.536812 -70.726712 20.135705 -16.727052
9.990000 0.250000 -7.350304 -0.607916 -82.138185 -77.959601 20.321566 -17.626298
9.995000 0.250000 -9.204748 -0.436030 -82.881255 -78.163145 21.723900 -17.480386
10.000000 0.250000 -10.570360 -0.150192 -83.541703 -75.363659 21.616698 -16.890325
```

Note

- Each registered tracker contributes one or more columns to the output file.
- Columns are ordered according to the registration sequence of trackers and fields.
- Missing or inactive drivers will result in zero-valued or unchanged outputs depending on the simulation state.
- The time column corresponds to the physical simulation time, not the iteration index.

4.6.9 Advanced Operators

4.6.9.1 Update Grid For RShape Driver

The purpose of this operator is to project the RShape (vertices, edges, and faces) onto the cells making up the exaDEM grid in order to speed up the search for interactions. Each grid cell is then assigned a set of vertices, edges, and faces that are potentially in contact with the cell's particles.

- Operator name: `grid_rshape_driver`
- Description: Update the list of information for each cell regarding the vertex, edge, and face indices in contact with the cell in a RShape driver.
- Parameters:

- *force_reset*: Force to rebuild grid.

Note

[1] This operator only projects the RShape driver onto the grid making up the MPI process subdomain. If the subdomain changes, the update must be forced (*force_reset*=0). [2] If the RShape is stationary (*v*= null, *vrot*=null), the grid is not updated. This speeds up calculations when the RShape driver has many elements.

YAML example:

```
- compute_driver_vertices:
  force_reset: true
```

4.6.9.2 Compute Driver Vertices

This operator is used to update the vertex positions of operators with vertices. For the moment, this operator is only used for RShape drivers and to fill in the *vertices* field.

- Operator name: `compute_driver_vertices`
- Description: This operator calculates new vertex positions.
- Parameters:
 - *force_host*: Force computations on the host

Note

For GPU performance reasons, you may decide not to update the GPU data directly, knowing that it will be used to build the CPU interaction list.

YAML example:

```
- compute_driver_vertices:
  force_host: false
```

4.6.9.3 Check Driver Displacement

This operator detects if a driver has moved more than 1/2 of the Verlet radius. This operator works in combination with the *backup_driver* operator to store driver data at the iteration when the interaction lists have been recalculated.

- In the case of a sphere, we test the distance between the two centers.
- In the case of a RShape, we check the displacement of all vertices.
- In the case of a cylinder, this option is disabled.
- In the case of a wall, we look at the difference between the offset values.

Currently, for the GPU version, these tests are carried out on the CPU, except for the detection of stl meshes, which requires a reduction operation. Operator characteristic:

- Operator name: `driver_displ_over`
- Description: It computes the distance between each particle in grid input and its backup position in *backup_dem* input. It sets result output to true if at least one particle has moved further than threshold.
- Parameters:

- threshold: Defined by the simulation (deduced from *rcut_inc*)

YAML example:

```
- driver_displ_over
```

4.7 Input/Output

In this section, we will describe the various operators used for analysis and I/O in DEM simulations.

4.7.1 Configuration

4.7.1.1 IO Configuration

This operator is used to define the tree structure of output files. By default, exaDEM proposes a tree structure with names defined in the *config_exaDEM* file. You can override this operator yourself by redefining it in the *io_tree* operator.

- Name: *io_config*
- Description: This operator defines the tree structure of output files.
- Parameters:
 - *avg_stress_tensor_name* : Write an output file containing stress tensors.
 - *dir_name* : Main output directory.
 - *interaction_basename* : Write an output file containing interactions.
 - *log_name* : Write an Output file containing log lines.

YAML example:

```
- io_config:  
  dir_name: "ExaDEMOutputDir"  
  log_name: "log.txt"  
  avg_stress_tensor_name: "AvgStressTensor.txt"  
  interaction_basename: "InteractionOutputDir-"
```

4.7.2 Checkpoint/Restart Operators

4.7.2.1 Reader Of xyz File

- Name: *read_xyz*
- Description: This operator reads a file written according to the xyz format.
- Parameters:
 - *bounds_mode* : default mode corresponds to ReadBoundsSelectionMode.
 - *enlarge_bounds* : Define a layer around the volume size in the xyz file. Default size is 0.
 - *file* : File name, this parameter is required.
 - *pbj_adjust_xform* : Adjust the form.

YAML example:

```
- read_xyz:
  file: input_file_rigid_surface.xyz
  bounds_mode: FILE
  enlarge_bounds: 1.0 m
```

How to build your xyz input file of *nb* particles.

```
nb
box_size_x box_size_y box_size_z
type_0 pos_x_0 pos_y_0 pos_z_0
type_1 pos_x_1 pos_y_1 pos_z_1
type_2 pos_x_2 pos_y_2 pos_z_2
...
type_nb-1 pos_x_nb-1 pos_y_nb-1 pos_z_nb-1
```

This is an example of two particles the same type in a domain $[[0,0,0],[10,10,10]]$.

```
2
10 10 10
0 2.5 5.0 5.0
0 7.5 5.0 5.0
```

4.7.2.2 Reader Of MPIIO File

- Name: *read_dump_particle_interaction*
- Description: This operator reads a dump file with all particle information required to restart the simulation. See operator: @write_dump_particles
- Parameters:
 - *filename* : Dump file name to read.
 - *bounds* : If set, override the domain's bounds, filtering out particles outside of overridden bounds (AABB = $[[\text{inf}_x, \text{inf}_y, \text{inf}_z], [\text{sup}_x, \text{sup}_y, \text{sup}_z]]$). If set to true and *shrink_to_fit* is true, try to reduce the domain's grid size to the minimum size enclosing fixed bounds.
 - *expandable* : If set, override domain expandability stored in file
 - *periodicity* : If set, overrides the domain's periodicity stored in a file with this value (ex: $[\text{false}, \text{true}, \text{false}]$).
 - *timestep* : If set, overrides the time iteration number.
 - *scale_cell_size* : If set, this option rescales cell size. Due to friction storage per cell, you can only multiply this size by an integer (1,2,4, 8, ...) or divide it by a power of $1/(2^n)$ (0.5,0.25 ...).
 - *shrink_to_fit* : If set to true and bounds were specified, try to reduce the domain's grid size to the minimum size enclosing fixed bounds.
 - *enable_extra_data* : Read Interactions.

YAML example:

```
- read_dump_particle_interaction:
  filename: last.dump
```

Note

This operator is used for spheres and not polyhedra because we need a special reader to read current interaction values containing the friction and moment. Show `read_dump_particle_interaction`.

4.7.2.3 Restart file script

In the `scripts` folder, you have the option of using a script `restart_template.py` that will enable you to quickly write the restart section of your simulation by directly retrieving the path to any saved files. This script directly includes whether the simulation mode is spherical or polyhedral. If the mode is polyhedral, the shape file will also be loaded automatically. The script will also check for a file containing driver data and include it. Finally, it will also include the last dump file (highest iteration).

Option:

- *directory*: Output file path, the default path is `ExaDEMOOutputDir`. Note that it should contain a subdirectory named: `CheckpointFiles` and this output file path is defined into the operator `io_tree`.

Example:

```
python3 ~/exaDEM/scripts/restart_template.py --directory SpheresMovableWallDir
```

Output:

```
Restart directory: SpheresMovableWallDir
Particle mode: Spheres
Last iteration identified: 29000
Here s a template for restarting your simulation at the last saved iteration:

includes:
- config_spheres.msp
- SpheresMovableWallDir/CheckpointFiles/driver_00000029000.msp

input_data:
- read_dump_particle_interaction:
    filename: SpheresMovableWallDir/CheckpointFiles/exadem_00000029000.dump
```

4.7.2.4 Reader Of Rockable Files

- Name: `read_conf_rockable`
- Description: This operator reads a rockable output file. However, some lines are not processed by the reader, such as the contact law type or its parameters.
- Parameters:
 - *bounds* This option defines the simulation domain. If not specified, the domain size is determined by the particle positions. Ex: `[[0,0,0],[1,1,1]]`.
 - *filename* Dump file name to read.
 - *enlarge_bounds* Define a layer around the volume size. Default size is 0.
 - ***bounds_mode* (str) Defines the bounding box calculation mode.**
Default is `COMPUTED_BOUNDS` (bounds are dynamically calculated from particles and shapes). An alternative is `DOMAIN` (uses predefined global domain limits).
 - *region* (*Region object*) Filters particles or IDs based on a specified geometric region.

- *vtk* (*bool*) If set to `true`, exports the shape geometry as VTK files for visualization.

Yaml Example:

```
input_data:
- read_conf_rockable:
  filename: input_file/518_poly.conf

input_data:
- read_conf_rockable:
  filename: input_a10.txt
  verbosity: false
  vtk: false
  bounds_mode: DOMAIN
  region: AREA
```

What is read:

- `t`
- `dt`
- `density`
- `periodicity`
- `nDriven`
- `shapeFile`
- `precision`
- `Particles`
- `Interactions` (read but NOT used)

What is not read:

- `Interactions` (read but NOT used)
- `Interfaces`
- `gravity`
- `AddOrRemoveInteractions`
- `UpdateNL`
- `forceLaw`
- `Contact law parameters`
- `interVerlet`
- `interConf`
- `tmax`
- `DVerlet`
- `dVerlet`
- `gravity`
- `ParamsInInterfaces`
- `dynamicUpdateNL`

- ContactPartnership
- cellMinSizes
- iconf
- boxForLinkCellsOpt

4.7.2.5 Read Shape File

The purpose of this operator is to add shapes to a collection of shapes. This operator can be called as many times as desired. However, if you add the same shape multiple times, it will create duplicates. Additionally, the shapes will be ordered according to the order of reading, meaning that type 0 will be associated with the first shape from the first input file. Furthermore, this operator will automatically create a polydata for each shape, which will be used for displaying the polyhedra using ParaView.

Note

The output Paraview file does not incorporate the “spherical” characteristics of polyhedra, i.e. surfaces are created by connecting the centers of vertices, edges are straight lines (instead of cylinders), and vertices are points (instead of spheres).

- ***read_shape_file*** :
 - *filename*: Input file name, no default name.
 - *scale_factor*: This option ‘scale_factor’ the input shapes. OBB, volume, vertices, and inertia are recomputed. Note that a vector of double should be provided. Example: *scale_factor*: [1.2,1,5.2].
 - *rename*: This option renames the input shapes. Note that a vector of string should be provided. Example: *rename*: [Shape1, Shape2, Shape3].
 - *rescale_minkowski*: This option disables the rescaling of the minkowski radius.
 - *verbosity*: It displays the typename and typeid.

Warnings:

Warning

- This operator takes on ASCII files.
- This operator is not typo-proof and will ignore problematic values.
- Do not define a shape “driven” such as a wall or a cylinder because the cell diameters and the cutoff radius for creating interaction lists are derived from the shapes of the polyhedra. These should be defined in specific operators if they have an analytical shape. If they have particular shapes with many facets, please use the RShape Driver reader (supported extension: *shp* or *stl*).

YAML example:

```
- read_shape_file:
  filename: shapes.shp

- read_shape_file:
  filename: shapes.shp
  rename: [PolyR, Octahedron]
```

(continues on next page)

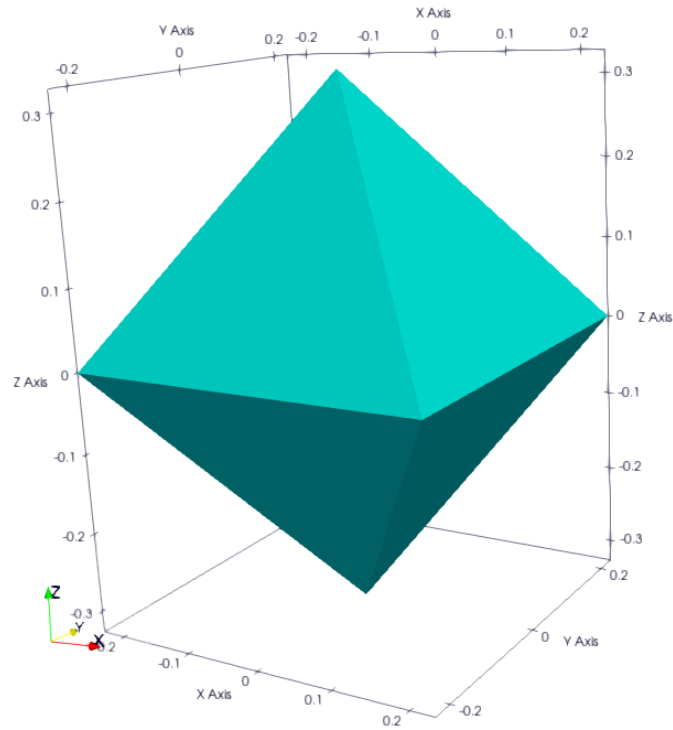
(continued from previous page)

```
- read_shape_file:
  filename: shapes.shp
  rename:   [ PolyRSize2, OctahedronSize2]
  scale_factor: [ 2.0, 2.0]
```

Example of a shape:

```
<
name Octahedron
radius 0.1
preCompDone y
nv 6
0.2310789034541148 -0.2310789034541148 0.0
0.2310789034541148 0.2310789034541148 0.0
0.0 0.0 0.32679491924311227
-0.2310789034541148 -0.2310789034541148 0.0
-0.2310789034541148 0.2310789034541148 0.0
0.0 0.0 -0.32679491924311227
ne 12
0 1
2 1
2 0
0 3
2 3
3 4
4 2
4 1
5 0
5 1
5 4
5 3
nf 8
3 0 1 2
3 2 3 4
3 1 2 4
3 0 2 3
3 0 5 1
3 0 5 3
3 3 5 4
3 4 5 1
obb.extent 0.33107890345411484 0.33107890345411484 0.4267949192431123
obb.e1 1.0 0.0 0.0
obb.e2 0.0 1.0 0.0
obb.e3 0.0 0.0 1.0
obb.center 0.0 0.0 0.0
position 0.0 0.0 0.0
orientation 1.0 0.0 0.0 0.0
volume 0.16666666666666666
I/m 0.04999999999999999 0.04999999999999999 0.04999999999999999
>
```

Example of *Octahedron.vtk* with paraview:



4.7.2.6 Writer Of MPIIO Files

- Name: *write_dump_particle_interaction*
- Description: This operator writes a dump file with all particle information required to restart the simulation. See operator: *@read_dump_particles*.
- Parameters:
 - *compression_level* Zlib compression level.
 - *filename* Dump output file name.
- Default behaviour: the default name is defined by : - *timestep_file*: “*exaDEM_%09d.dump* and piloted by *simulation_dump_frequency*: 1 in the operator *global*.

Note

This operator is defined in the default *ExaDEM* operator named *dump_data_particles*.

4.7.2.7 Writer Of Rockable Files

- Name: *write_conf_rockable*
- Description: This operator writes a minimal rockable output file to use the rockable tool *see*.
- No parameter

YAML example:

```
iteration_dump_writer:
- write_conf_rockable
```

Note

Only particles data are written to use the rockable tool *see*.

4.7.2.8 Writer Of XYZ Files

- Name: `write_xyz`
- Description: This operator writes a txt file (.xyz) with all specified fields.
- Parameters: `*fields`: array of fieldsets. Example: [`id`, `velocity`, `radius`] `*filename`: name of the output file. `*units`: array of units. Example: { `velocity`: "m/s", `radius`: "m" }

Note

The first line of the output file contains the number of particles. The second line contains the “lattice” description, useful when using ovito.

YAML example: Replaces MPIIO output files with xyz files.

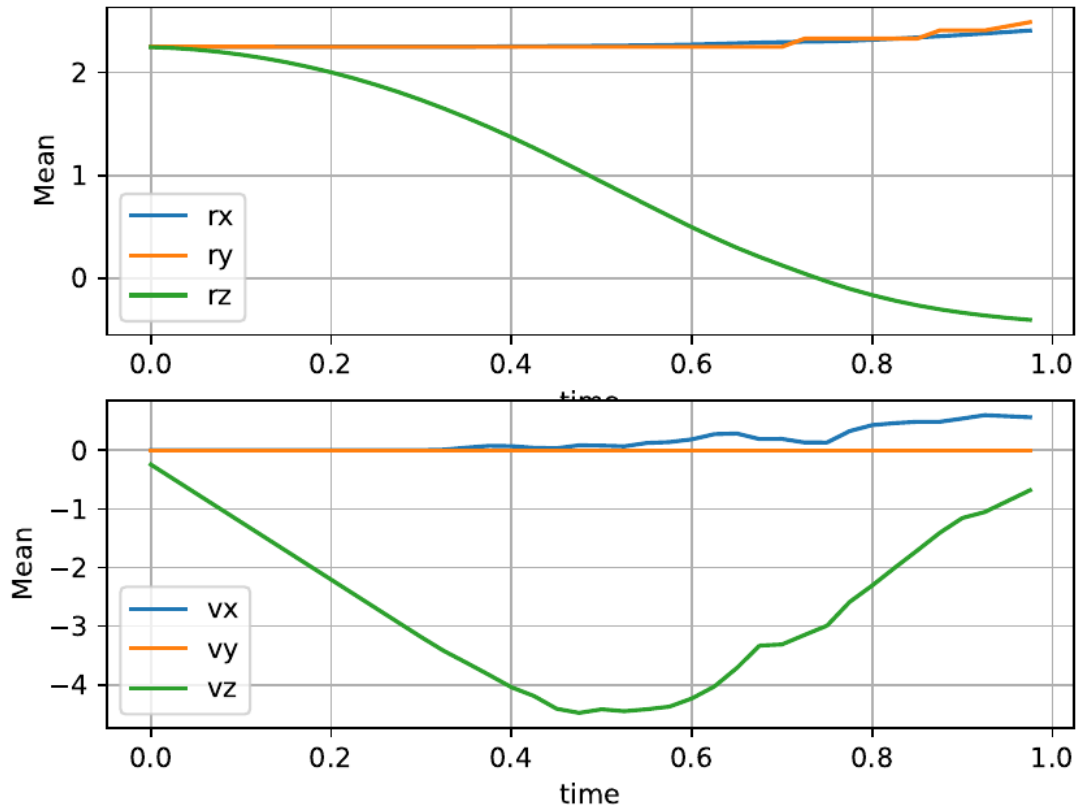
```
dump_data_xyz:
- timestep_file: "dem_pos_vel_%09d.xyz"
- write_xyz:
  fields: [ id, velocity, radius ]
  units: { velocity: "m/s", radius: "m" }

iteration_dump_writer:
- dump_data_xyz

global:
  simulation_dump_frequency: 500
```

To process these files, a sample script is provided in `scripts/post_processing/profile_pos_vel.py`. This is a minimal, easily modifiable post-processing file that calculates the averages of all position and velocity components.

Output file: [mean_r_v.pdf]



4.8 Post Processings

In this section, we will describe the operators related to the usage of polyhedra or spheres.

4.8.1 Paraview For Polyhedra

In exaDEM, there are two ways to display polyhedra with Paraview:

- The first is to directly display the vertices of the polyhedra and the surfaces in parallel VTP (PolyData). However, no fields associated with the polyhedra are available, such as velocity or density.
- The second solution is to use the generic Paraview output of *exaDEM* by adding the orientation field and the homothety field (optional). Then, it's possible to associate a mesh with each point, such as an *octahedron.vtk* file generated by *read_shape_file*, to each point by associating it with a size (*field::homothety*) and a quaternion (*field::orient*).

Note

Only the default behavior when the *config_polyhedra.msp* file is used is the option 2 that offers more possibilities. In addition, it is important to note that paraview does not include the layer of shape->radius size, i.e. faces are displayed according to the vertex centers.

- **Option 1: *write_paraview_generic***

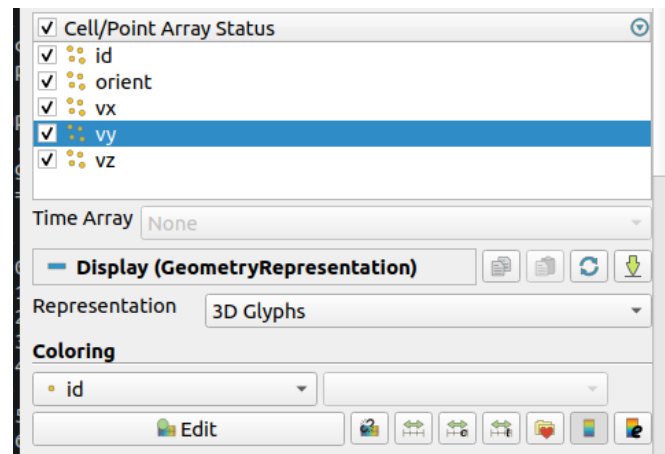
- *binary_mode* [BOOL] : paraview format file, default is true.
- *compression* [STRING] : level of compression, default is “default” for vtkZLibDataCompressor.
- *filename* [STRING]: basename of the parallel paraview output files, default is “output”.
- *write_ghost* [BOOL]: dump ghost particles, default is false.
- *write_box* [BOOL]: write box information in a box.vtp file, default is true.
- *write_external_box* [BOOL]: write external box (ghost area), default is false.
- This operator is based on this function: *ParaviewWriteTools::write_particles*.

YAML example:

```
write_paraview_generic:
  binary: false
  write_ghost: false
  fields: ["vx", "vy", "vz", "id", "orient"]
```

How to use it with Paraview:

- Firstly, we need to load our reference mesh (Octahedron.vtk) in our case.
- Secondly, load your particle file into the Paraview folder (default name in exaDEM for the paraview_generic operator).
- Thirdly, choose the 3D Glyphs representation and the coloring.



- Fourthly, in the Glyph Parameters section, choose “Orient” with the orientation mode “Quaternion” and as orientation vectors: “orient”. To change the size, you can check Scaling and add the Scale Array you wish. Finally, in the Glyph Type dropdown menu, select “Pipeline Connection” and in Input, choose “Octahedron.vtk”.

Glyph Parameters

☒ Orient

Orientation Mode Quaternion

Orientation Vectors orient

☐ Scaling

Scale Mode No Data Scaling Off

Scale Factor x ↺

Scale Array None

Glyph Type Pipeline Connection

Input Octahedron.vtk

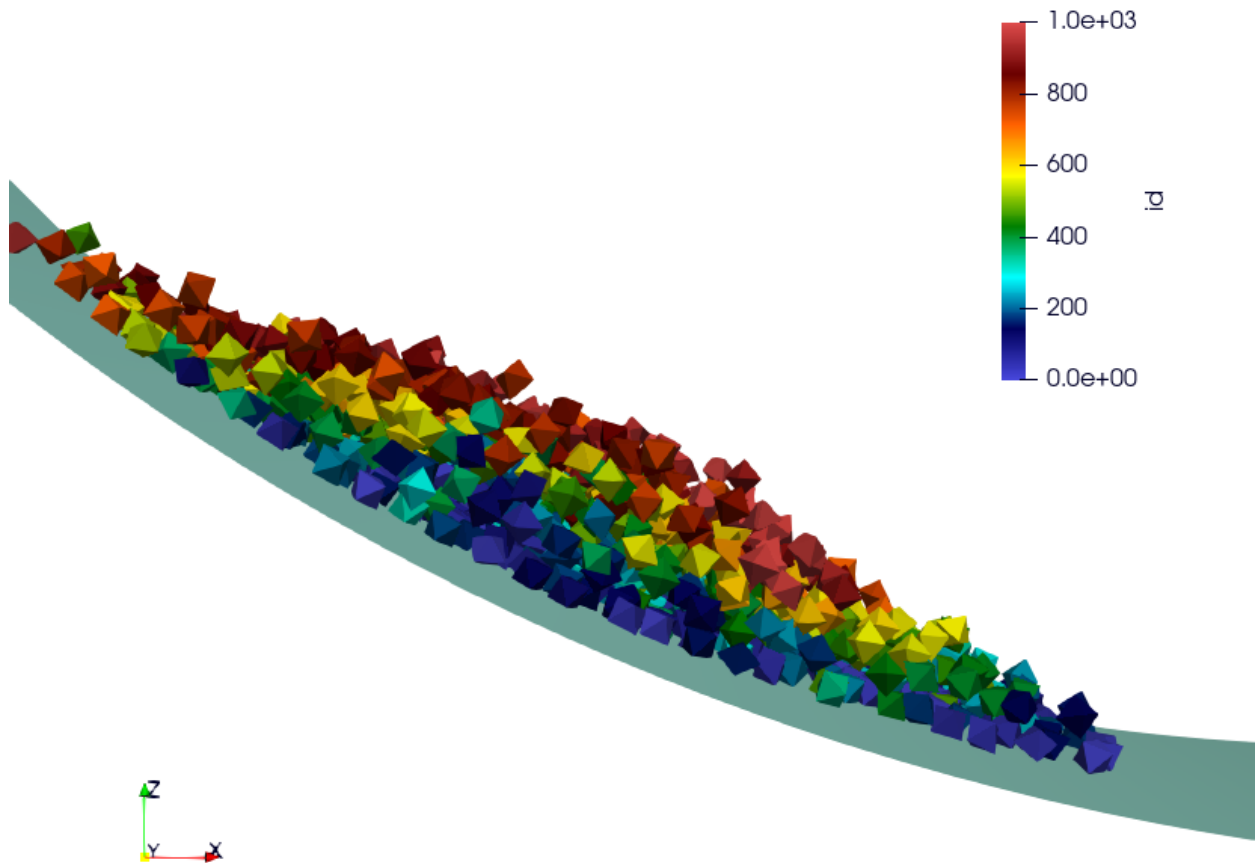
☒ Use Glyph Table

Glyph Table Index Array None

☐ Use Composite Glyph Table

☐ Use Glyph Culling And LOD

Result for a simulation of 1000 Octahedra falling in a cylinder colored by their ID:



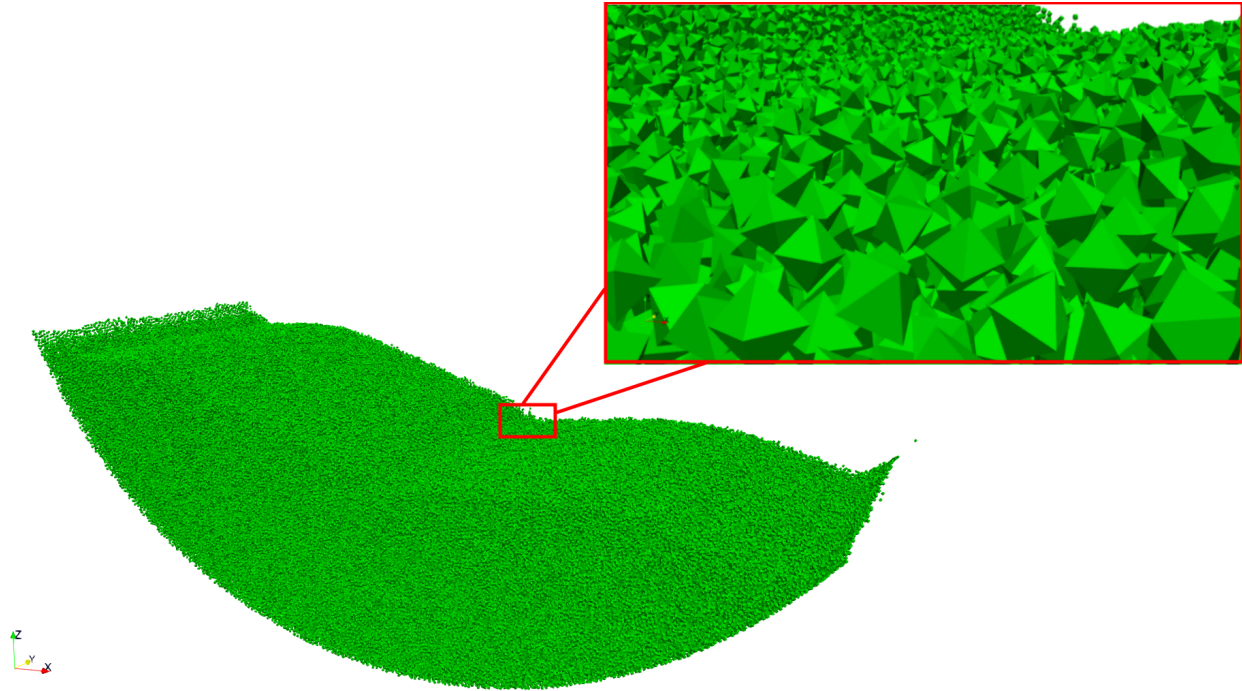
- **Option 2: write_paraview_polyhedra**

- *filename* : Name of paraview file, there is no default name. Note that in *ExaDEM*, filename is defined in the default execution stream.
- *mpi_rank* : Add a field containing the mpi rank. [optional]

YAML example:

```
particle_write_paraview_generic:
- write_paraview_polyhedra:
  mpi_rank: true
```

Example with 850,000 octahedra:



Note

This operator is rather limited in terms of visualization, so we now advise you to use option 1, which offers more possibilities (field display) and less memory-intensive files.

4.8.2 Paraview With OBB

This operator allows you to display OBBs around polyhedra in paraview. These files are stored in different files from those used to store polyhedron information. By default, these files are available in the directory *ExaDEMOutput-Dir/ParaviewOutputFiles/* under the format *obb_%010d.pvtp*. The fields associated with OBBs are the polyhedron ID and type.

- ***write_paraview_obb_particle:***

- *basedir* : Name of the directory where paraview files will be written
- *basename* : Name of paraview file, there is no default name. Default is “obb”.
- *timestep* : Current simulation time is defined.

Note

This operator is called after *write_paraview_generic* and is triggered by *simulation_paraview_frequency* called into the global operator.

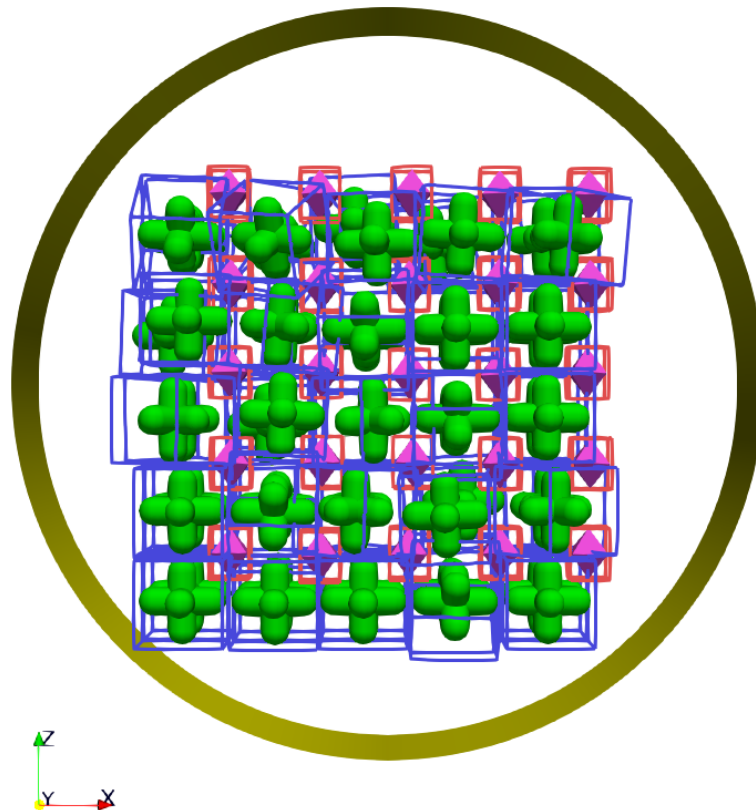
Warning

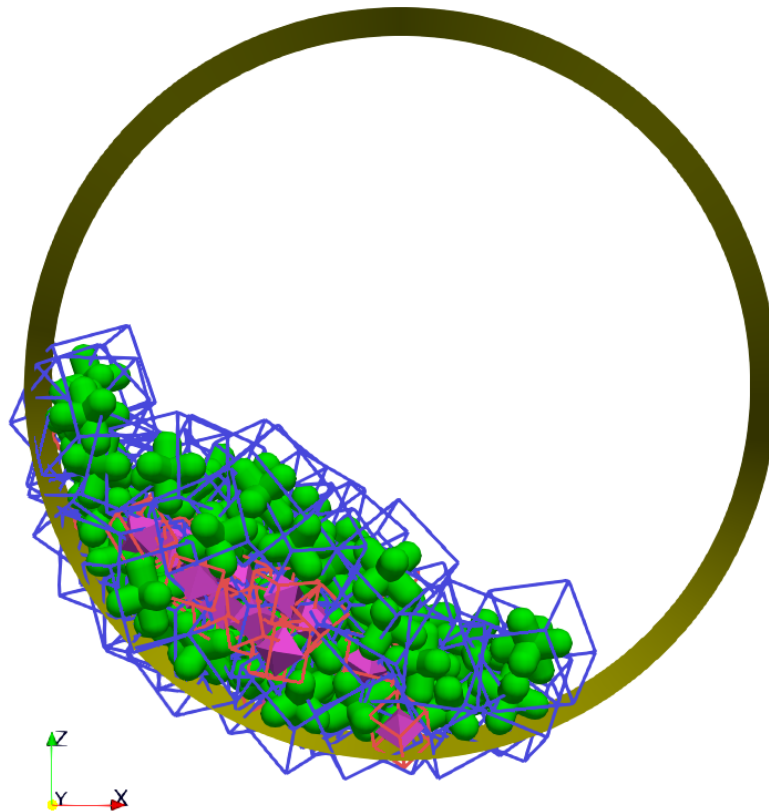
This operator doesn't work for simulations with spheres.

YAML example:

```
- write_paraview_obb_particle
```

Output example:





4.8.3 Contact Network

This operator is used to visualize the contact network between polyhedra using ParaView. For each active contact/interaction, we assign the value of the normal force calculated in Contact's law. You can enable this option, which will be automatically triggered at the same time as the other paraview files, with the option `enable_contact_network: true` in global. See examples: "Polyhedra/Example 2: Octahedra in a Rotating Drum" and "Spheres/Example 1: Rotating drum".

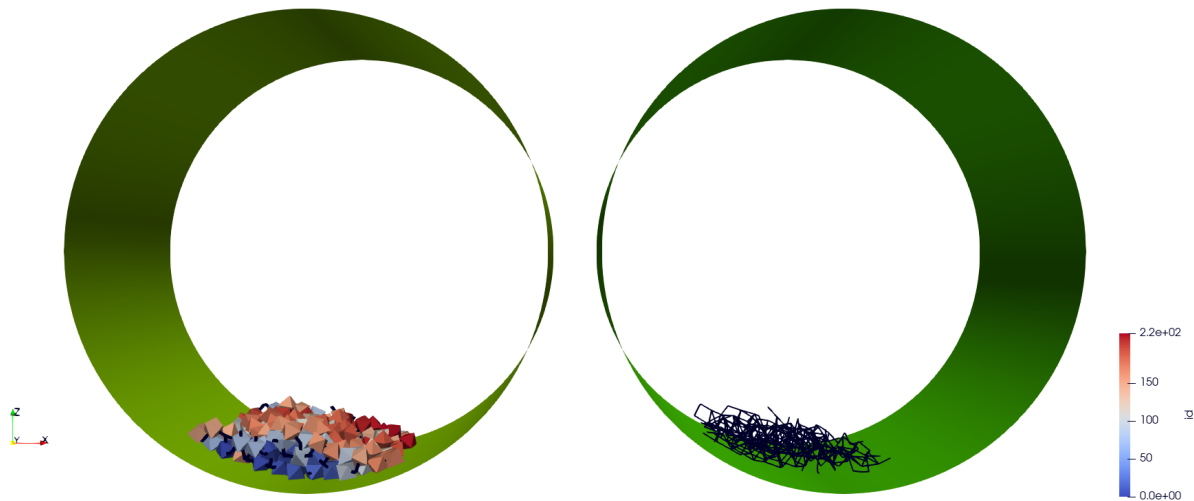
- **`dump_contact_network:`**
 - `filename` : Name of the paraview file, there is no default name.
 - `timestep` : Current simulation time is defined.

YAML example:

```
- timestep_paraview_file: "ParaviewOutputFiles/contact_network_%010d"
- dump_contact_network
```

```
global:
  enable_contact_network: true
```

Here is an example of 216 polyhedra after a fall into a cylinder, left the simulation and right the contact network:



Comments / Extensions:

- This operator can be modified to display more values per contact. To achieve this, you need to change the type of *StorageType* in the *NetworkFunctor* structure. Then, you'll need to populate this function in the operator (*exaDEM::Interaction* I, const size_t offset, const size_t size*). Finally, you'll need to add a field in *write_pvt* and include this field in *write_vtp*.
- Currently, this operator doesn't take particularly long to execute and isn't called frequently. However, it doesn't benefit from any shared-memory parallelization (OpenMP) because the network storage is implemented using a *std::map*.

4.8.4 Project Field Quantities onto Cartesian Grid

This operator projects the sum of one or several field values onto a Cartesian grid corresponding to the grid defined by the cells of the DEM mesh. It is then possible to use the *write_grid_vtklegacy* operator to generate a Paraview output.

Operator description:

- Name: *quantities_cell_projection*
- Parameters:
 - *fields* [list of strings]: List of regular expressions to select fields to project. Example: *fields*: [vx, ay, stress].
 - *grid_subdiv* [double]: Number of subdivisions applied to each grid cell. Default is 1, i.e no subdivision.
 - *splat_size* [int >= 1]: Smoothing factor controlling the spatial influence of each projected value. Default is 1.0.

to use this operator, the simplest way is to define the analysis frequency (all) in the global operator (*simulation_analyses_frequency*) and add the *quantities_cell_projection* operator to the operator analyses, as in the following example (see *example/polyhedron/analysis/particle_cell_projection.msp*):

YAML example: available here *example/polyhedron/analysis/particle_cell_projection.msp*

```
global:
  simulation_analyses_frequency: 1000

analyses:
  - timestep_paraview_file: "ParaviewOutputFiles/quantities_%010d.vtk"
  - message: { mesg: "Write " , endl: false }
```

(continues on next page)

(continued from previous page)

```

- print_dump_file:
  rebind: { mesg: filename }
  body:
    - message: { endl: true }
- resize_grid_cell_values
- quantities_cell_projection:
  splat_size: 0.99
  grid_subdiv: 2
  fields: [count, vnorm, vx, vy, vz, stress]
- write_grid_vtklegacy

```

Paraview output:

In this figure, we show:

- Left: the relative velocity (surface plot),
- Middle: a glyph representation of the grid using arrows,
- Right: a zoomed-in view showing the octahedra.

ExaDEM provides several pre-configured cell projection examples in `data/config/config_cell_projection.msp`:

- `all_cell_proj`:
 - description: Projects all available fields.
 - output: `ParaviewOutputFiles/quantities_XXX.vtk`
- `vnorm_cell_proj`:
 - description: Projects the sum of particle velocity norms in each cell.
 - output: `ParaviewOutputFiles/vnorm_XXX.vtk`
- `velocity_cell_proj`:
 - description: Projects particle velocity magnitude and components (`vx`, `vy`, `vz`).
 - output: `ParaviewOutputFiles/velocity_proj_XXX.vtk`
- `stress_cell_proj`:
 - description: Projects the stress field.
 - output: `ParaviewOutputFiles/stress_proj_XXX.vtk`
- `count_cell_proj`:
 - description: Counts the number of particles per cell.
 - output: `ParaviewOutputFiles/count_proj_%010d.vtk`

Note

For all of these post-processing operators, the `count` output is also added to allow computing averages of projected quantities. These operators don't use the `subgrid` option.

YAML example:

Full source code: `example/polyhedra/analyses/particle_cell_projection_count_vnorm_v_stress.msp`

```

global:
  simulation_analyses_frequency: 1000

analyses:
  - vnorm_cell_proj
  - count_cell_proj
  - stress_cell_proj
  - velocity_cell_proj

```

Paraview outputs:

4.8.5 Dump Interaction Data

This feature outputs the main information for each interaction, for more information as interaction type see *Polyhedra - Interaction / Contact* or *Spheres - Interaction / Contact*. This feature has been implemented to enable post-simulation analysis. An option has been added to the `contact_polyhedron` and `contact_sphere` operators to output interaction data as a CSV file. To activate it, simply modify the value of `analysis_interaction_dump_frequency` in the operator block `global`.

Output files are located in the *ExaDEMOutputDir/ExaDEMANalysis* folder. For each iteration (XXX) with file writing, a folder containing an interaction file is created, such as: *Interaction_XXX/Interaction_XXX_MPIRANK.txt*.

For each interaction, we write:

- The particle identifier `i` [uint64_t],
- The particle identifier `j` or driver identifier [if type in [4,12]] [uint64_t],
- The sub-identifier of the particle `i` [int],
- The sub-identifier of the particle `j` [int],
- The interaction type [int <= 13],
- The deflection / overlap [double <= 0, > 0 if contact type = “cohesion”],
- The contact position [Vec3d],
- The normal force [Vec3d],
- The tangential force [Vec3d],
- Particle position `i` [Vec3d],
- Particle position `j` [Vec3d], [0,0,0] if it is a driver.

Warning

Inactive interactions have been filtered out when writing output files. In addition, symmetrized interactions are stored one time.

Note

An example is available in: `example/polyhedra/analyses/interaction.msp`

ExaDEM also offers post-processing scripts for basic interaction analyses. The scripts can be used as a basis for developing other analyses according to need. The first available script is *interaction_summary.py* :

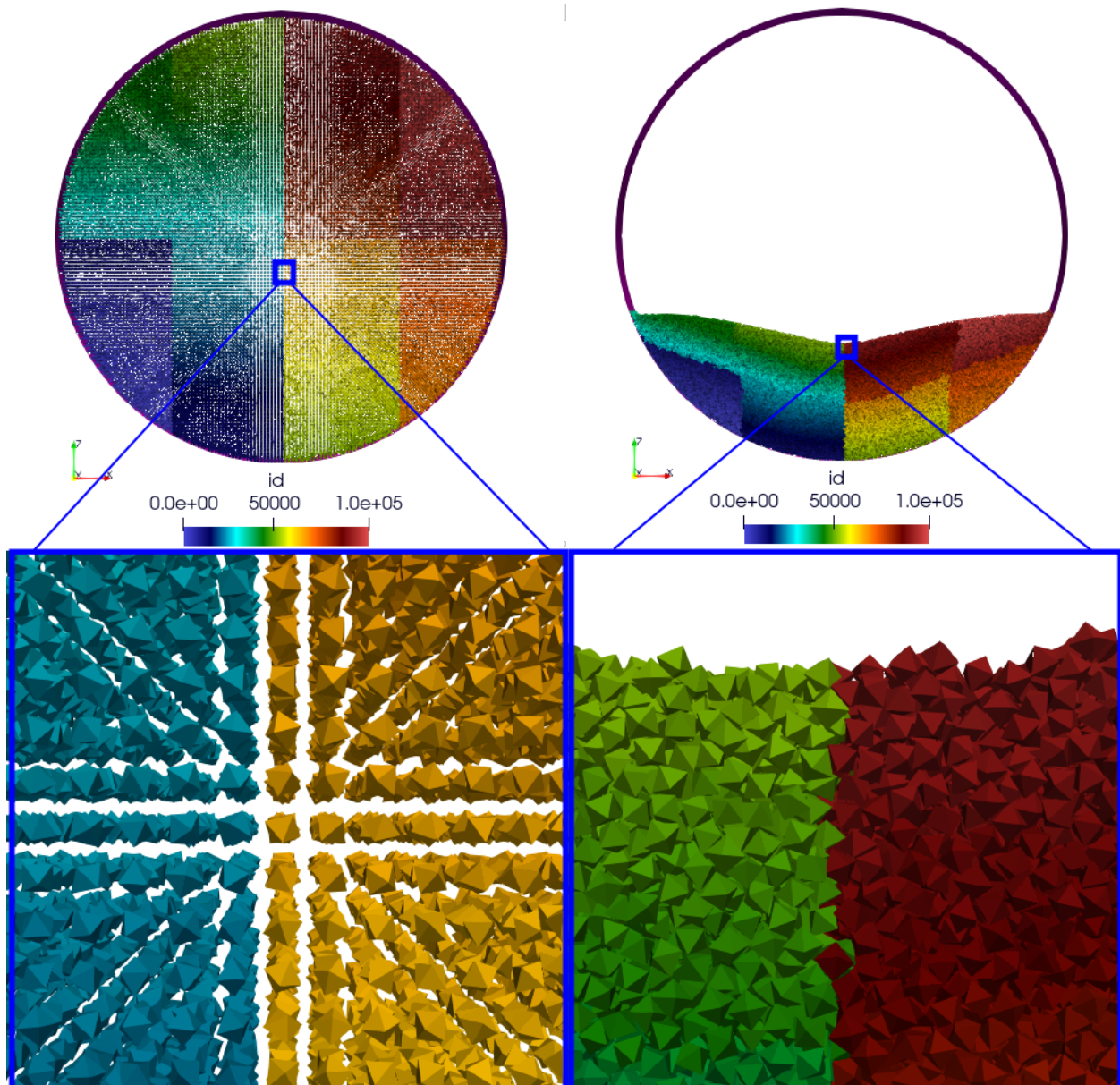
- Read all interaction files
- Plot the number of interactions per type as a function of the timestep (*types.pdf*)
- Plot the number of interactions as a function of the timestep (*count.pdf*)

How to run this script:

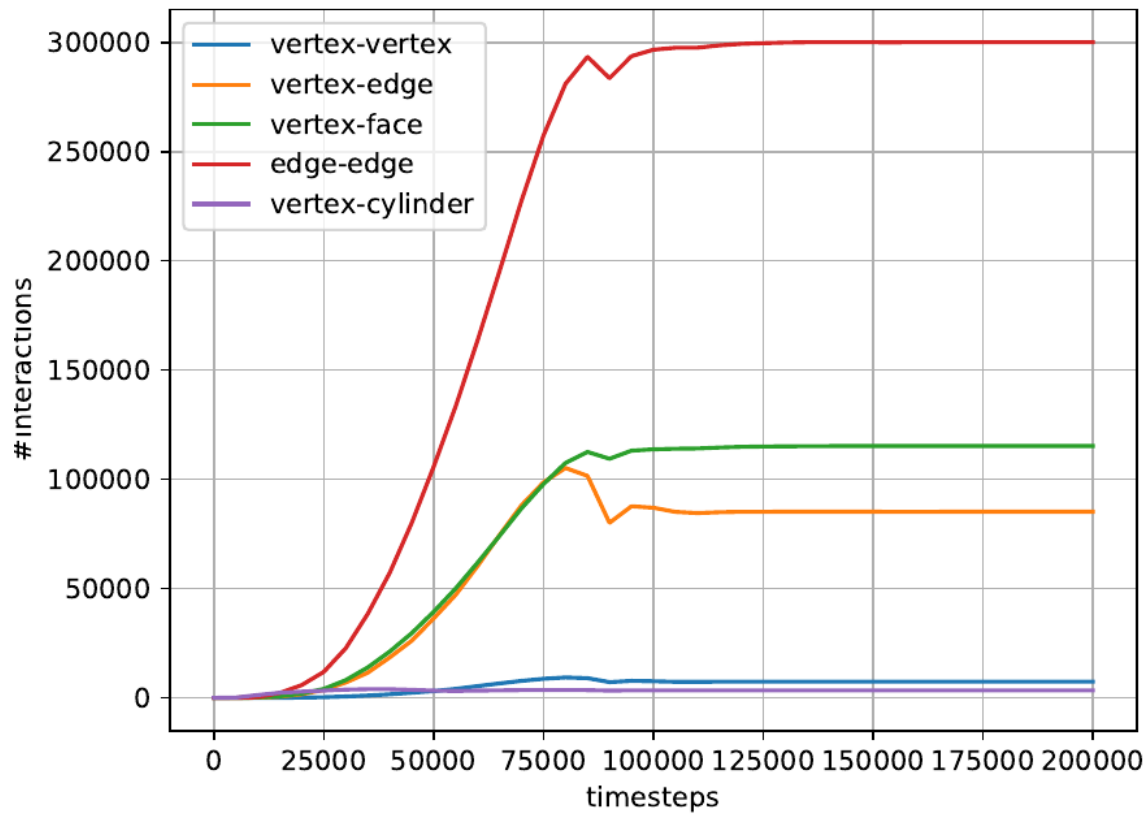
```
cd ExaDEMOutputDir/ExaDEMANalyses
python3 PATH_TO_ExaDEM/scripts/post_processing/interaction_summary.py
```

Output file examples:

Simulation: near 104,000 octahedral particles over 200,000 timesteps of $5 \cdot 10^{-5}$ s falling into a cylinder.



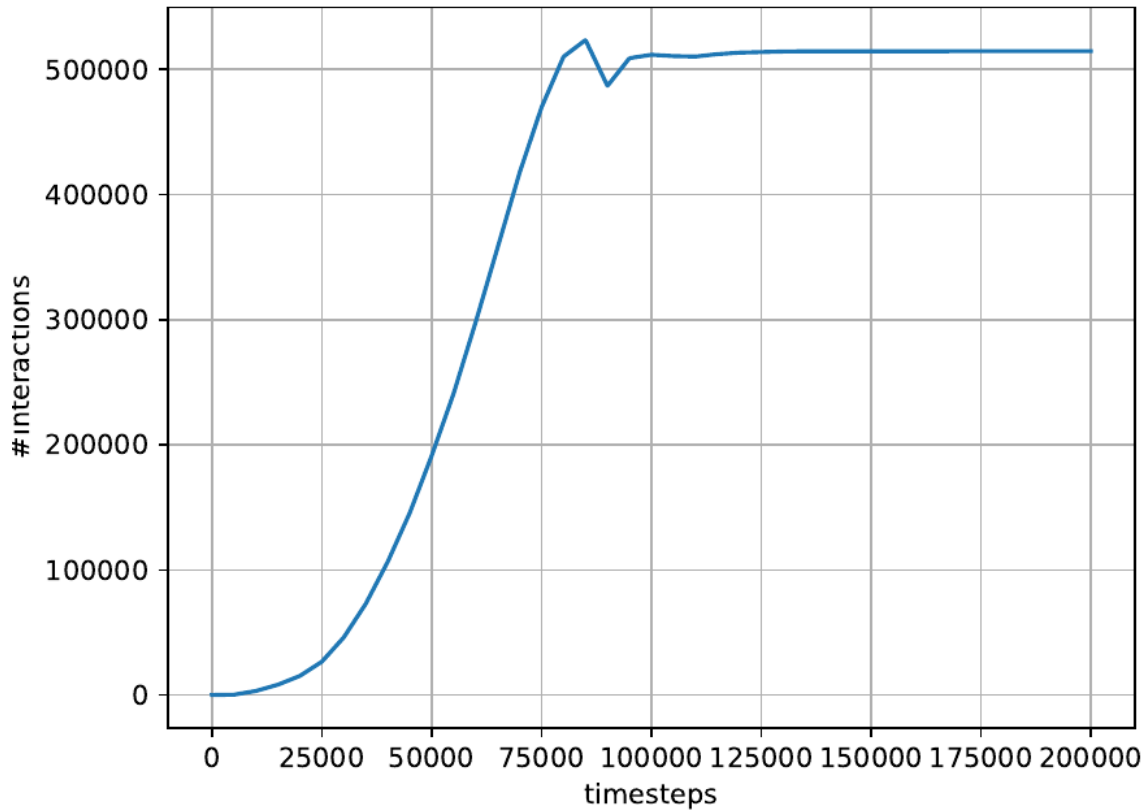
- types.pdf



Note

Symmetrized interactions are counted twice within the `interaction_summary.py` python script.

- count.pdf



4.8.6 Interaction Summary

This operator allows displaying the total number of interactions, both total and active. An interaction is considered active if there is contact and, consequently, if the cumulative friction is different from $\text{Vec3d}\{0,0,0\}$. It also enables the separation of different types of interactions: Vertex-Vertex, Vertex-Edge, Vertex-Face, and Edge-Edge.

- Name: *stats_interactions*
- No parameter.
- Tip: Add this operator when performing recurring but infrequent operations such as Paraview outputs or check-point output files (see YAML example).

YAML example:

```
+dump_data_paraview:
- stats_interactions
```

Output example:

```
=====
* Type of interaction      = active / total / ghost
* Number of interactions   = 369404 / 369404 / 0
* Vertex - Vertex         = 41623 / 41623 / 0
* Vertex - Edge           = 85161 / 85161 / 0
```

(continues on next page)

(continued from previous page)

```

* Vertex - Face           = 21997 / 21997 / 0
* Edge   - Edge           = 23922 / 23922 / 0
* Vertex - Cylinder       = 0 / 0 / 0
* Vertex - Surface        = 3770 / 3770 / 0
* Vertex - Ball           = 0 / 0 / 0
* Vertex - Vertex (Driver) = 0 / 0 / 0
* Vertex - Edge (Driver)  = 0 / 0 / 0
* Vertex - Face (Driver)  = 0 / 0 / 0
* Edge   - Edge (Driver)  = 0 / 0 / 0
* Edge (Driver) - Vertex  = 0 / 0 / 0
* Face (Driver) - Vertex  = 0 / 0 / 0
* Vertex - Vertex (Stick) = 192931 / 192931 / 0
=====

```

4.8.7 Global Stress Tensor

A stress tensor for a given particle is computed such as:

$$\sigma_{loc} = \sum_{ij \in I} [f_{ij} c_{ij}^T]$$

With I the active interactions, $f_{ij} = (f_{x_{ij}}, f_{y_{ij}}, f_{z_{ij}})$ the forces between the particle i and j , and $c_{ij} = r_i - p_{ij}$ the vector between the center of the particle i noted r_i and the contact position of the interaction I named p_{ij} .

And the total stress of the system : $\sigma = \frac{1}{V} \sum_{loc} [\sigma_{loc}]$, with V the volume.

Stress tensor calculation is performed by the `stress_tensor` operator, and writing to a .txt output file is performed by the `write_stres_tensor` operator. To trigger the writing of the stress tensor, simply declare the `analysis_dump_stress_tensor_frequency` variable to the frequency chosen in the global operator of your YAML file (.msp), which by default is set to -1.

YAML example:

```

global:
  simulation_end_iteration: 150000
  simulation_log_frequency: 1000
  simulation_paraview_frequency: 10000
  analysis_dump_stress_tensor_frequency: 1000

```

For further information

This frequency triggers several things. When passing through the `Contact Force` operator, the list of interactions / normal force / tangential force is stored in the classifier. The stress tensor is then calculated in `global_stress_tensor` and written in `write_stress_tensor`. By default, volume is calculated from the simulation volume using the `compute_volume` operator. So, by default, the frequency will trigger the chaining of these three operators:

```

compute_volume:
  - domain_volume

global_stress_tensor:
  - average_stress_tensor

dump_stress_tensor_if_triggered:
  condition: trigger_write_stress_tensor
  body:

```

(continues on next page)

(continued from previous page)

```
- compute_volume
- global_stress_tensor
- write_stress_tensor
```

In the case of a particle deposit or other simulation where the simulation domain does not correspond to the simulation volume, you can either implement your `my_volume` operator and replace the `compute_volume` operator block such as:

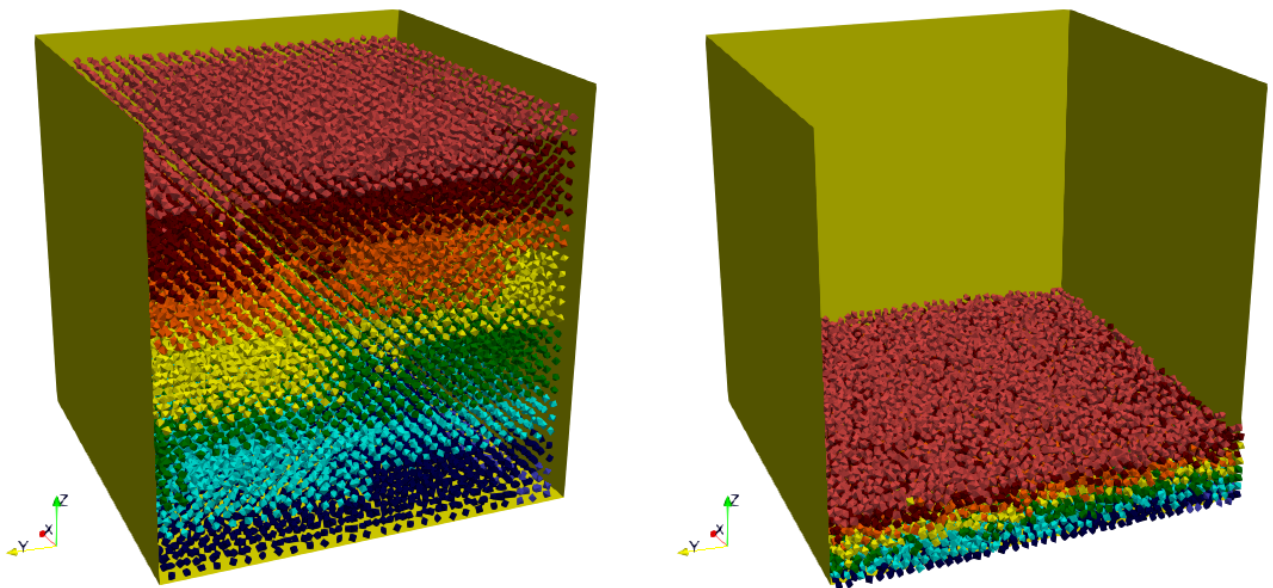
```
compute_volume:
- my_volume:
  my_param1: x
  my_param2: y
  my_param3: z
```

If you want to directly assign the value of a fixed-size volume, we advise you to add these lines to your input file:

```
compute_volume: nop

global_stress_tensor:
- average_stress_tensor:
  volume: 21952
```

A usage example is available at the following address: example/polyhedra/analyses/write_avg_stress.msp. It involves dropping a set of hexapods into a box and watching the stress tensor evolve over time.



A gnuplot script is available at scripts/post_processing/avg_stress.gnu to quickly plot lines:

```
set key autotitle columnhead
N = system("awk 'NR==1{print NF}' AvgStressTensor.txt")
```

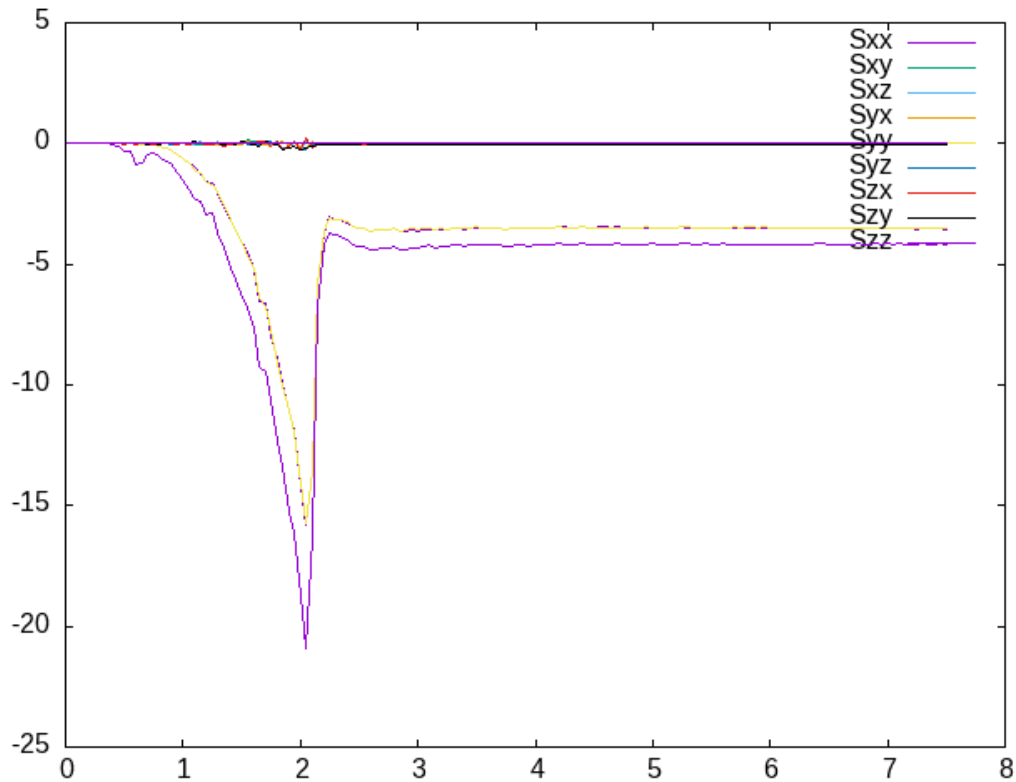
(continues on next page)

(continued from previous page)

```

plot for [i=2:N] "AvgStressTensor.txt" u 1:i w l
set key autotitle columnhead
set term png
set output "avgStress.png"
replot

```



4.8.8 Particle Counter

The purpose of this operator is to count the number of particles per type in a particular region.

Operator:

- Name: `particle_counter`
- Parameters:
 - *name*: Filename. Default is: `ParticleCounter.txt`,
 - *types*: List of particle types (required, `[0,1,2, ...]`),
 - *region*: Choose the region, default is the domain simulation.

To use this operator, the simplest way is to define the analysis frequency (all) in the global operator (`simulation_analyses_frequency`) and add the `particle_count` operator to the operator analyses, as in the following example (see `example/polyhedron/analysis/particle_counter.msp`):

```

global:
  simulation_analyses_frequency: 100000

analyses:

```

(continues on next page)

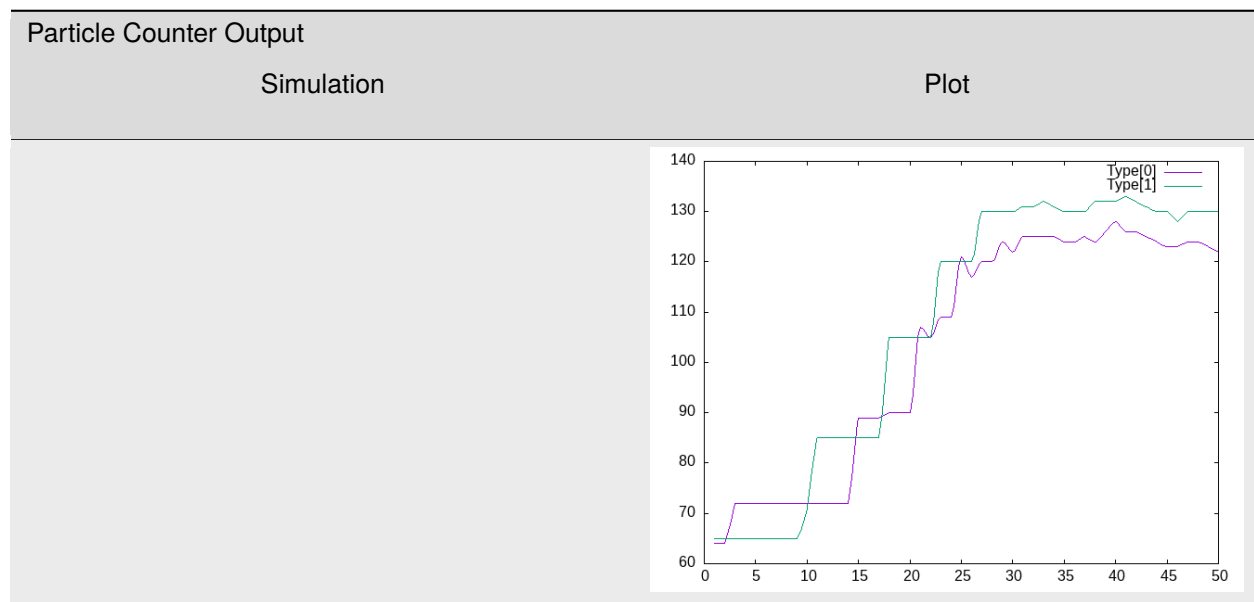
(continued from previous page)

```
- particle_counter:
  name: "ParticleTypes0And1.txt"
  types: [0,1]
  region: BOX
```

One possibility for post-processing is to use gnuplot with the following commands:

```
set key autotitle columnhead
set style data lines
plot for [i=2:3] 'ExaDEMOutputDir/ExaDEMANalyses/ParticleTypes0And1.txt' using 1:i_
↪smooth mcsplines
```

Results:



4.8.9 Barycenter

The purpose of this operator is to count the number of particles per type in a particular region.

Operator:

- Name: `particle_barycenter`
- Parameters:
 - *name*: Filename. Default is: `ParticleBarycenter.txt`,
 - *types*: List of particle types (required, `[0,1,2, ...]`),
 - *region*: Choose the region, default is the domain simulation.

To use this operator, the simplest way is to define the analysis frequency (all) in the global operator (`simulation_analyses_frequency`) and add the `particle_count` operator to the operator analyses, as in the following example (see `example/polyhedron/analysis/particle_barycenter.msp`):

```
global:
  simulation_analyses_frequency: 10000
```

(continues on next page)

(continued from previous page)

```

analyses:
- particle_barycenter:
  name: BaraycenterBox.txt
  types: [0,1]
  region: BOX
- particle_barycenter:
  name: Baraycenter.txt
  types: [0,1]

```

One possibility for post-processing is to use gnuplot with the following commands:

```

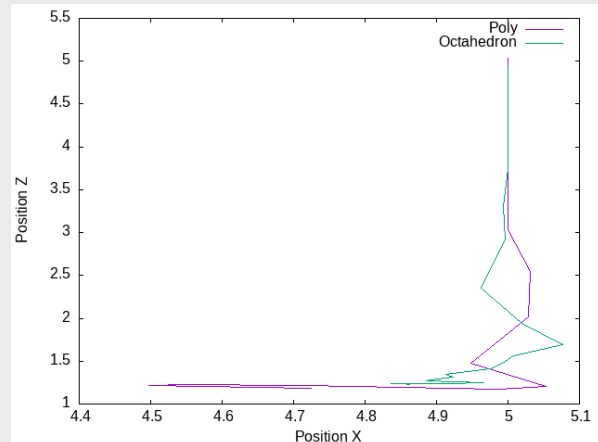
set key autotitle columnheade
set style data lines
set title font "Barycenter per particle type"
set xlabel "Position X"
set ylabel "Position Z"
plot "PolyhedraAnalysisBarycenterDir/ExaDEMANalyses/Baraycenter.txt" u 2:4 w l title
  ↳ "Poly"
replot "PolyhedraAnalysisBarycenterDir/ExaDEMANalyses/Baraycenter.txt" u 5:7 w l title
  ↳ "Octahedron"
set terminal png
set output "barycenter_plot.png"
replot

```

Particle Counter Output

Simulation

Plot



4.9 External Packages

4.9.1 RSA_MPI package

RSA MPI is an HPC library dedicated to generating random configurations of non-intersecting balls, with an unbiased RSA algorithm. The sequential strategy is based on Ebeida et al, 2012. This strategy has been extended to the MPI framework, as described in this paper “Parallel and bias-free RSA algorithm for~maximal Poisson-sphere sampling,

Josien & Prat (in preparation)”.

- Operator name: `rsa_rnd_rad`
- Parameters:
 - *enlarge_bounds*: Add an empty layer around your domain. Default is 0.0.
 - *periodicity*: if set, overrides domain’s periodicity.
 - *bounds*: Override domain’s bounds, filtering out particles outside of overridden bounds.
 - *type* : Define particle, default is 0.
 - *expandable* : if set, override domain expandability.
 - *r_min* : Value of the smallest radius possible.
 - *r_max* : Value of the largest radius possible.
 - *n_max* : Maximal number of particles. Default is *1e16*.
 - *region* : The region where the sample is generated.

YAML examples:

```
- rsa_rnd_rad:  
  bounds: [[ -4.97 , -4.97 , 1 ] , [ 4.97 , 4.97 , 150 ] ]  
  r_min: 0.04  
  r_max: 0.05  
  n_max: 60000000  
  region: MYSUPERZONE  
  type: 0
```

- Operator name: `rsa_vol_frac`
- Description: This operator initializes a grid with particles using the RSA (Random Sequential Adsorption) MPI-based algorithm. It allows the user to control particle generation via geometric constraints, volume fractions, and particle type definitions. This is intended for use only at simulation initialization.
- **Parameters:**
 - *particle_type_map*: Mapping between particle type names and their internal identifiers.
 - *enlarge_bounds*: Add an empty layer around the domain. Default is 0.0.
 - *bounds_mode*: Controls how bounds are interpreted: from file or overridden manually.
 - *periodicity*: If set, overrides domain’s periodicity.
 - *expandable*: If set, overrides domain expandability.
 - *bounds*: Override domain bounds. Particles outside of these bounds will be filtered out.
 - *params*: List of RSA particle parameters. Each entry defines [radius, volume fraction, type name].
 - *region*: Optional Constructive Solid Geometry (CSG) region used to restrict particle placement.
 - *use_shape*: This option uses the shape data to fill information like the ‘radius’. Please do not use it with spheres.

YAML examples:

For spheres:


```

- particle_type:
  type: [ Sphere0 , Sphere1 , Sphere2 ]
- rsa_vol_frac:
  periodic: [true,true,false]
  bounds: [ [ 10 , 10 , 10 ] , [ 14, 14, 14 ] ]
  params: [[0.5, 0.1, Sphere2], [0.25, 0.1, Sphere1], [0.125, 0.1, Sphere0]]
- set_fields:
  polyhedra: false
  type: [ Sphere2 , Sphere1 , Sphere0 ]
  radius: [ 0.5 , 0.25 , 0.125 ]
  velocity: [ [0,0,0] , [0,0,0] , [0,0,0] ]
  sigma_velocity: [ 0.001 , 0.001 , 0.001 ]
  density: [ 0.02 , 0.02 , 0.02 ]

```

For polyhedra:

```

- read_shape_file:
  filename: shapes.shp
  rename: [PolyR, Octahedron]
- rsa_vol_frac:
  bounds: [ [0 , 0 , 0 ] , [40 , 10 , 40 ] ]
  params: [ {vf: 0.055, type: PolyR},
            {vf: 0.055, type: Octahedron}]
  use_shape: true
- set_fields:
  polyhedra: true
  type: [ PolyR, Octahedron ]
  velocity: [[0,0,0], [ 0,0,0 ] ]
  sigma_velocity: [ 0.1, 0.1 ]
  random_quaternion: [ true, true ]

```

Note

The domain defined by bounds must match a multiple of the cell size defined in the domain operator. This operator does not require `init_rcb_grid` to be set.

See *Example 6: Rescale shape* .

DEVELOPER GUIDE

This project is written in C++-20.

5.1 ExaDEM Changelog

All notable changes to this project will be documented in this file.

5.1.1 Release Note

5.1.1.1 Release Notes v1.2.3 (in development)

Note

This release is under development on the `main` branch and has not been tagged yet.

New Features:

- Fields
 - Add a `group` particle field, used to select multi-material contact parameters independently from the particle type. Several particle types can share the same group and therefore reuse the same contact law.
 - Add the `group` parameter to `set_fields`.
 - Add the `set_group` operator to (re)assign the `group` field to particles according to their type.
- Force Field
 - Add the `check_group_completeness` operator, which stops the simulation if the `group` particle field references a group index not covered by `multimat_contact_params/drivers_contact_params`.
- I/O
 - Add support for the `group` field in dump file readers/writers, including legacy Rockable readers (legacy writers are not affected).

Changes and Enhancements:

- Force Field
 - `multimat_contact_params`, `drivers_contact_params` and `inner_bond_params` now select contact parameters using `group1/group2` (or `group`) instead of particle type names (`mat1/mat2/mat`). See the Multi-Material section of the Force Field page.

5.1.1.2 Release Notes v1.2.2 (07/26)

New Features:

- Fragmentation / Inner Bonds
 - Add operator to randomly change sphere radius
 - Add inner bonds and driver support to compute the stress per particle
 - Add criterion mode (mixed and separated)
 - Add `dump_broken_interface` and a ParaView post-processing script
 - Add stress energy criterion
 - Add inner bonds in the dump contact network
- Visualization / ParaView
 - Improve `ParaviewNetwork` and the criterion used when writing interactions
 - Add cluster field for the `paraview_polyhedra` operator
 - Add ParaView output for cylinders
 - Add ParaView export in pvtu format
- Drivers
 - Rework `driver_params` and `MotionType` in the driver plugin
 - Use a dt scale in driver numerical schemes
 - Add a `driverextractor`
 - Fix driver surface corrections on GPU

Changes and Enhancements:

- Performance
 - Integrate the interaction list build step
 - Optimize `unclassify` with binary search
 - Reduce compilation time for contact polyhedron/sphere
 - Use a dedicated extra storage for `rshape` data
 - Use scratch memory for `nbh_polyhedron_gpu`
- Cleanup and Refactoring
 - Clean the drivers plugin
 - Move `classify/unclassify` implementation to `.cpp` files
 - Move interaction manager implementation to `.cpp` files
 - Rename `interaction_buffers` to `contact_state`
 - Clean the shape plugin
 - New convention for structure members: `name` -> `name_`
- CI / Build
 - Add a CUDA CI pipeline
 - Use CUDA 13 in CUDA CI

Bug Fixes:

- Fix ghost interactions handling in `nbh_polyhedron_gpu`
- Fix ill-formed interactions in `nbh` operators
- Fix compilation in debug mode
- Update `grid_cell_values` on `ghost_update`
- Fix use of MEM PREFETCH with CUDA 13
- Fix logs and add a face shape identification tolerance
- Fix periodic boundary conditions with `DOMAIN_BOUNDS` mode (rockable)

5.1.1.3 Release Notes v1.2.1 (03/26)**New Features:**

- Add a `deposit` plugin.
- Add an operator to add particles from an XYZ file.

Changes and Enhancements:

- I/O
 - Switch from `grid_vtklegacy` to `grid_vtk` for grid output.
- Fragmentation
 - Add support for weights in inner bond force computations.
- General
 - Add an additional validation check in `check_rcut`.

5.1.1.4 Release Notes v1.2.0 (02/26)**New Features:**

- Fragmentation System
 - Add inner bond interaction (rework of the interaction system)
 - Add interfaces (set of inner bond interactions)
 - Add a routine to stick polyhedra
 - Add field `cluster`
 - Add `config_fragmentation.msp`
- Shapes
 - Add a default shapes library (`cube`, `sphere`, `rice grain`)
- Drivers
 - Add a deformation operator for STL meshes
 - Rename/refactor `Stl_mesh` to `RShapeDriver`
 - Refactor `FORCE_MOTION` into a more versatile `PARTICLE` motion type
 - Add a “Freeze Particles” operator
 - Improve expression-based motion types to account for angular moments

Changes and Enhancements:

- Performance
 - Copy interactions in the ghost area (MPI)
 - Compute the contact interface criterion on GPU
 - Update BackupDEM structures for more efficient checkpointing
- Post-Processing
 - Add a traction/shear test case for fragmentation validation
 - Improve ParaView outputs for particle interfaces
 - Fix vertex ordering in interface exports

Bug Fixes:

- Fix compatibility with ExaNBody 2.0.7
- Fix region usage in `set_fields`
- Improve radius-from-shape detection logic

5.1.1.5 Release Notes v1.1.6 (12/25)

New Features:

- Add TinyExpr support and projected quantities
- Add a motion modification operator
- Enable STL driver control using moments
- Add a script to visualize driver-applied stresses by component
- Improve ParaView output for contact networks
- Add a DMT enumeration

Bug Fixes:

- Fix multiple Rockable behavior issues
- Fix ParaView export bugs for polyhedra and vertex displacement
- Fix simulation box definition and the Rockable dump writer
- Fix an MPI bug in linear compressible motion
- Add validation for a missing `preCompDone` flag in shape files

5.1.1.6 Release Notes v1.1.5 (10/25)

Bug Fixes:

- Fix `linear_compressive_forces` and `linear_motion` modes when moving a surface in the negative direction.

Added Examples:

- Spheres with linear motion types
- Polyhedra with linear compressive force

5.1.1.7 Release Notes v1.1.4 (09/25)

New Features:

- Add a pendulum motion type for the Surface driver
- Add the `check_consistency_grid_classifier` operator
- Add a rescale Minkowski option in `read_shapes`
- Implement `exadem-config` file generation

Changes and Enhancements:

- Examples
 - Update the axial-stress example for spheres
 - Update rotating-drum examples
 - Refactor `polyhedra/rotating_drum`, `polyhedra/stl_mesh` and `polyhedra/balls`
 - Refactor `ball/driver` examples and fix neighbor list update
 - Refactor the `spheres/movable-wall` example
 - Update and factorize analysis examples
 - Remove deprecated examples under `depot`
- Regression Tests
 - Add regression tests for `sphere/cylinder_stl`, mirror boundary conditions, `polyhedra/defbox`, `polyhedra/stl_cylinder`, `spheres/piston` and `funnel`

5.1.1.8 Release Notes v1.1.3 (09/25)

New Features:

- Add a mixer simulation example
- Add examples with large vertex sets
- Integrate OBBTree (from Rockable) for improved collision detection
- Add shape rescaling at import time

Changes and Enhancements:

- Improve required-slot handling
- Centralize and factorize error management (`color_log::error`, `color_log::warning`)

Bug Fixes:

- Fix ParaView export of STL meshes
- Fix MSP driver output files

5.1.1.9 Release Notes v1.1.2 (07/25)

New Features:

- Add support for dumping Rockable object output files
- Add a new internal vertex data layout for improved efficiency
- Add customizable options to `compute_cell_particles`

- Add a Shaker motion type for STL mesh drivers (sinusoidal displacement)

Bug Fixes:

- Fix ParaView shape data output

5.1.1.10 Release Notes v1.1.1 (05/25)

New Features:

- Add the `check_rcut` operator to verify the interaction cutoff radius.
- Add a multi-material contact law supporting multiple parameter sets.
- Add an RSA-based particle generator targeting a given volume fraction.
- Add a slot to override domain bounds when importing geometry with `read_xyz`.
- Add logging for the critical time step.

5.1.1.11 Release Notes v1.1.0 (03/25)

New Features:

- Fields
 - Add Unified operator for initializing fields: `set_fields`
- Drivers
 - Add driver motion type:
 - * `STATIONARY`
 - * `LINEAR_MOTION`
 - * `COMPRESSIVE_FORCE`
 - * `LINEAR_FORCE_MOTION`
 - * `FORCE_MOTION`
 - * `LINEAR_COMPRESSIVE_MOTION`
 - * `TABULATED`
 - Add a reader for binary stl files
- Analysis
 - Add new analyses:
 - * Compute barycenters per type
 - * Count the number of particles per type

Changes and Enhancements:

- I/O
 - Optimized the contact network (add information)
- Update deformation domain: available for spheres and polyhedra

5.1.1.12 Release Notes v1.0.2 (11/24)

New Features

- Drivers
 - Added moving and rotating drivers, with special support for stl_mesh drivers.
 - Introduced timestep operators.
 - Implemented I/O driver operators.
- I/O
 - Defined a tree structure for output files (io_config).
 - Computed system stress tensor.
 - Added example for writing to XYZ files.
 - Included post-processing scripts for easier data analysis.

Changes and Enhancements

- Drivers
 - Optimized stl_mesh to efficiently build the lookup grid.
- Classifier for Interactions
 - Improved interaction storage in the classifier (InteractionWrapper) using SOA (Structure of Arrays) data structures.
- I/O
 - Added new logs to track active interactions and maximum interpenetration.
- General
 - Renamed hooke_force operator to contact_force for clarity.
 - Limited browsing to cells containing particles only.

Bug Fixes

- Fixed shape name paths to be relative.
- Updated interaction statistics to print the number of interactions per driver.

5.1.1.13 Release Notes for Version 1.0.1 (06/24):

New Features:

- Classifier for Interactions
 - Introduced a classifier to categorize interactions and enhance data organization.
 - Provides improved management and processing of interaction data.
- Mirror Boundary Conditions from ExaNBody Are Available
 - Mirror boundary conditions to simulate reflections at domain boundaries.
- GPU Version for Polyhedra
 - Added GPU-accelerated support for polyhedra computations.

Changes and Enhancements:

- Unified Drivers for Spheres and Polyhedra

- Integrated a unified driver system for both spherical and polyhedral objects.
- Simplifies driver management and enhances code reusability.

Removed Features:

- Meshset and Friction Plugins
 - Removed deprecated meshset and friction plugins.
 - Reduces complexity.

Added Examples:

- New Example: Funnel
- Mirror Boundary Conditions

5.2 Parallelization

5.2.1 MPI Parallelization

5.2.1.1 Domain Decomposition

5.2.1.2 Cost Model

The weight associated with each cell used for load distribution using the RCB method is as follows:

$$W_i = 1 + 3.P_i + \sum_{j \in 0, \dots, N_i} [F(I_i[j].type)]$$

with W_i the array of weights, i is the cell id, I_i the interactions in the cell i , P_i the number of particles in the cell i , N_i is the number of interactions, and F is the cost function associated to the interaction type.

Table 1: F values

Value	Type	F(Value)
0	Vertex - Vertex	1
1	Vertex - Edge	3
2	Vertex - Face	5
3	Edge - Edge	4
4	Vertex - Cylinder	1
5	Vertex - Surface	1
6	Vertex - Ball	1
7	Vertex - Vertex (RShape Driver)	1
8	Vertex - Edge (RShape Driver)	3
9	Vertex - Face (RShape Driver)	5
10	Edge - Edge (RShape Driver)	4
11	Vertex (RShape Driver) - Edge	3
12	Vertex (RShape Driver) - Face	5

5.2.2 Thread Parallelization

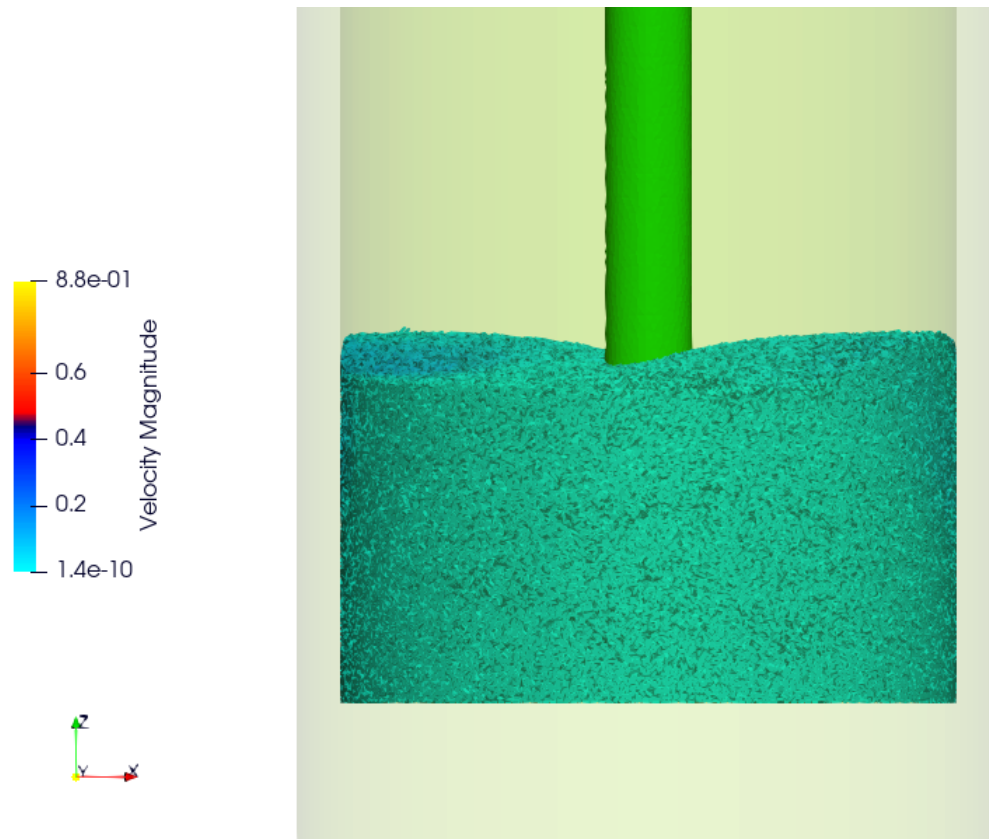
5.2.3 GPU Support

5.2.4 Benchmarks

5.2.4.1 Blade - CPU (Polyhedron)

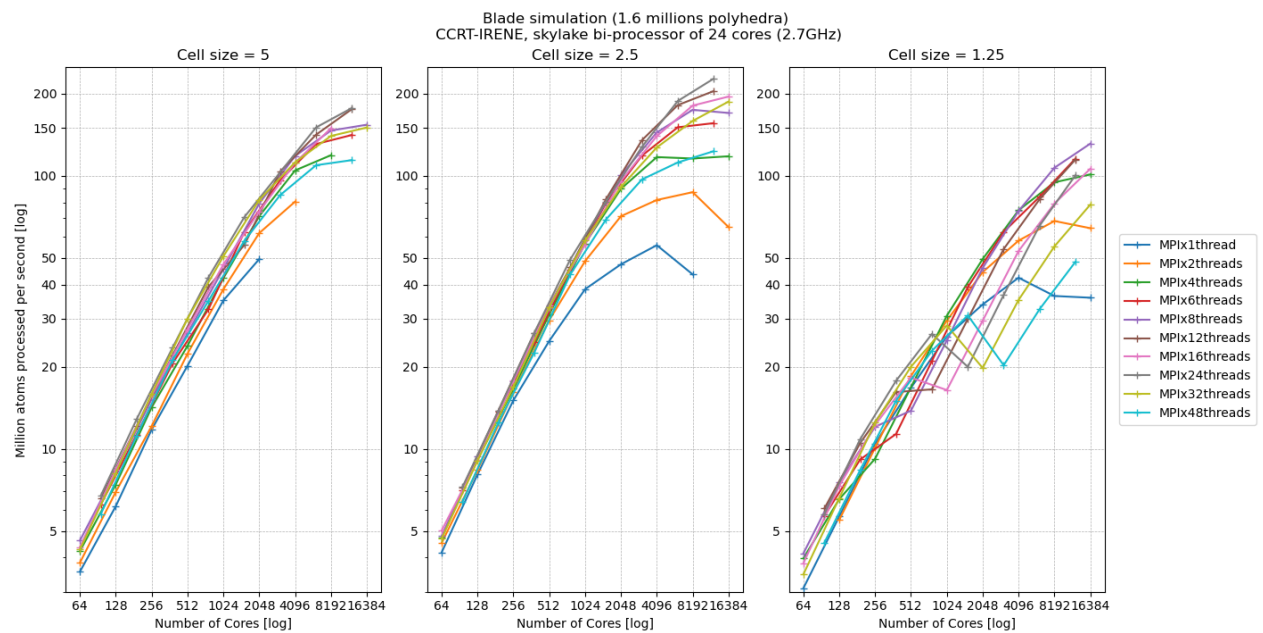
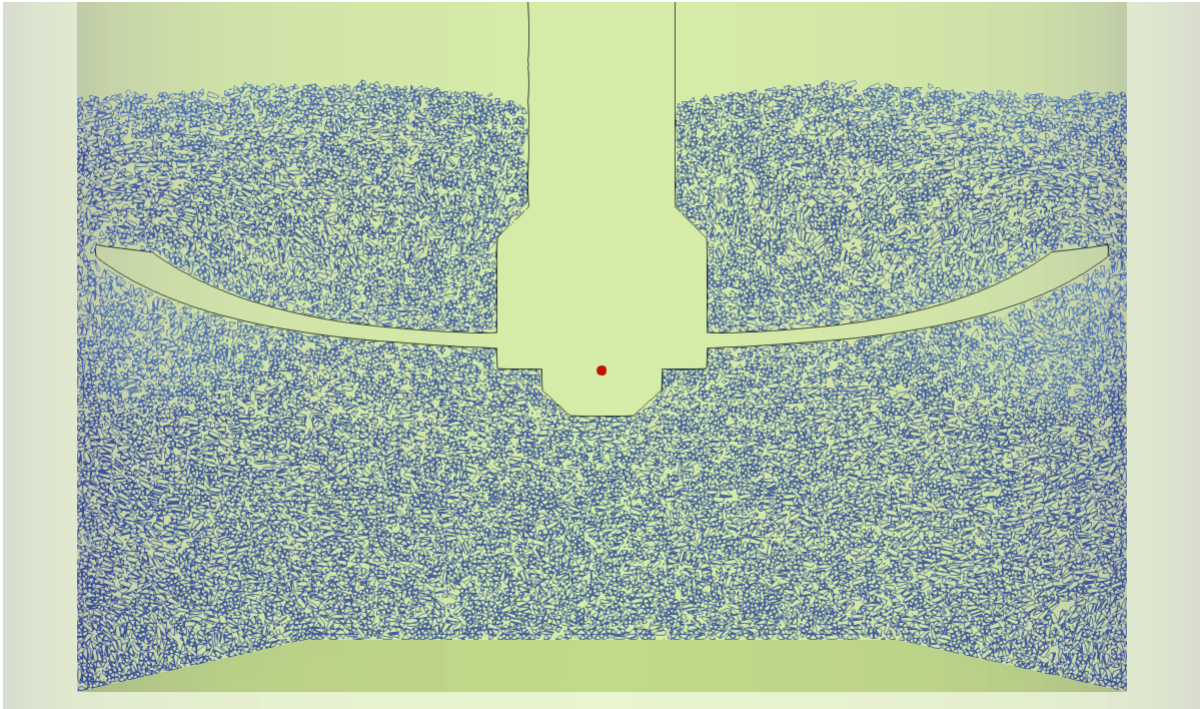
Information:

- Blade: {25,952 faces, 77,856 edges, 28,497 vertices}
- Number of timesteps: 10,000
- Number of polyhedra: 1,651,637
- This benchmark is achieved by reproducing the tutorial simulation (Blade) with polyhedra 5 times smaller. See images below.
- Test the hybrid parallelization MPI + OpenMP
- Machine used: CCRT-IRENE with:
 - Node: bi-processor of skylake
 - Each processor has 24 cores
 - Frequency: 2.7GHz
 - until 16,384 cores



Some remarks:

- For this test, the simulation domain is split into 39,000 cells and then distributed among the MPI processes. Some configurations fail because there are not enough cells per MPI process.
- Best results are obtained with 24 OMP threads per MPI process. This corresponds to the number of cores per socket. NUMA effects are likely to reduce performance beyond this.



5.2.4.2 Rotating Drum - GPU (Polyhedron)

This example is defined in the repository: <https://github.com/Collab4exaNBody/exaDEM-benchmark/tree/main/rotating-drum-poly> . This simulation is run on an a100 GPU using 32 cores. Result format: Loop time (Update Particles/Force Field).

Table 2: GPU Benchmarks

Version	Case 10k	Case 80K
v1.0.1 (06/24)	28.1 (17.1 / 6.8)	71.6 (37.8 / 26.0)
v1.0.2 (11/24)	23.3 (17.7 / 4.1)	48.9 (33.0 / 13.8)
master (05/12/24)	6.38 (2.61 / 2.49)	17.6 (11.65 / 4.3)
v1.1.0 (27/03/25)	6.99 (2.43 / 2.56)	17.3 (10.94 / 4.4)
v1.1.2 (22/07/25)	10.4 (1.99 / 5.37)	16.59 (9.35 / 4.2)
v1.1.3 (01/09/25)	10.77 (1.9 / 5.7)	16.23 (9.0 / 4.1)
v1.1.4 (24/09/25)	8.36 (3.5 / 2.4)	18.0 (10.8 / 4.1)
v1.1.5 (26/10/25)	8.44 (3.3 / 2.6)	18.0 (10.8 / 4.1)

5.2.4.3 Rotating Drum - CPU (Sphere)

This benchmark has been presented in 2023 within the paper [1] : “ExaNBody : a HPC framework for N-Body applications”.

Note

We are no longer able to reproduce this performance as the code has changed (hopefully for the better), but it shows the performance of the first prototype. The input files are no longer available either, as the operators have changed and the data structures have changed, notably with the improvements for interaction and parallelization per interaction instead of per particle in *OpenMP/gpu*.

Description:

Different OpenMP/Mpi configurations (number of cores/threads per mpi process) have been tested to balance multi-level parallelism. Both simulations were instrumented during 1,000 representative iterations. The performance of ExaDEM was evaluated using up to 256 cluster nodes, built on bi-socket 64-core AMD EPYC Milan 7763 processors running at 2.45 GHz and equipped with 256 GB of RAM.

ExaDEM’s performance is evaluated with a simulation of a rotating drum containing 100 million spherical particles, see the figure below. This setup is a tough benchmark as particles are rapidly moving all around the heterogeneously dense domain due to gravity. Additionally, the employed contact force model has a low arithmetic intensity, and exaDEM must handle pairwise friction information that is updated by kernel and must migrate between mpi processes when subdomains are redistributed.

Note

Note that the Milan nodes are made up of 128 cores spread over 8 NUMA nodes, and we have pointed out that NUMA effects reduce overall performance.

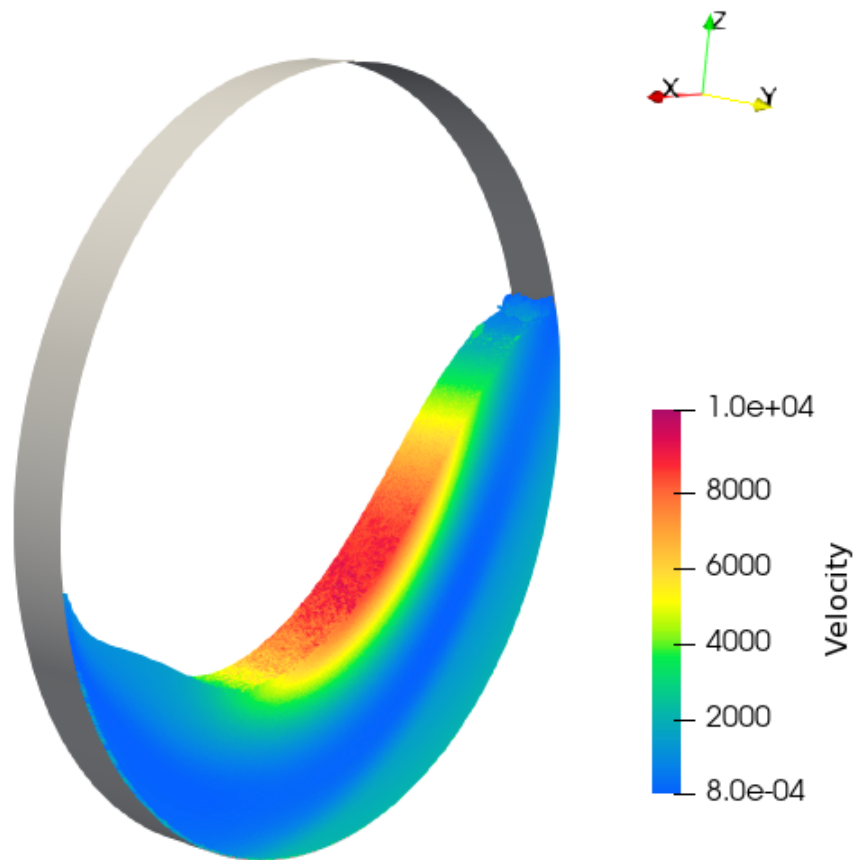


Fig. 1: DEM simulation of 100 million spheres in a rotating drum.

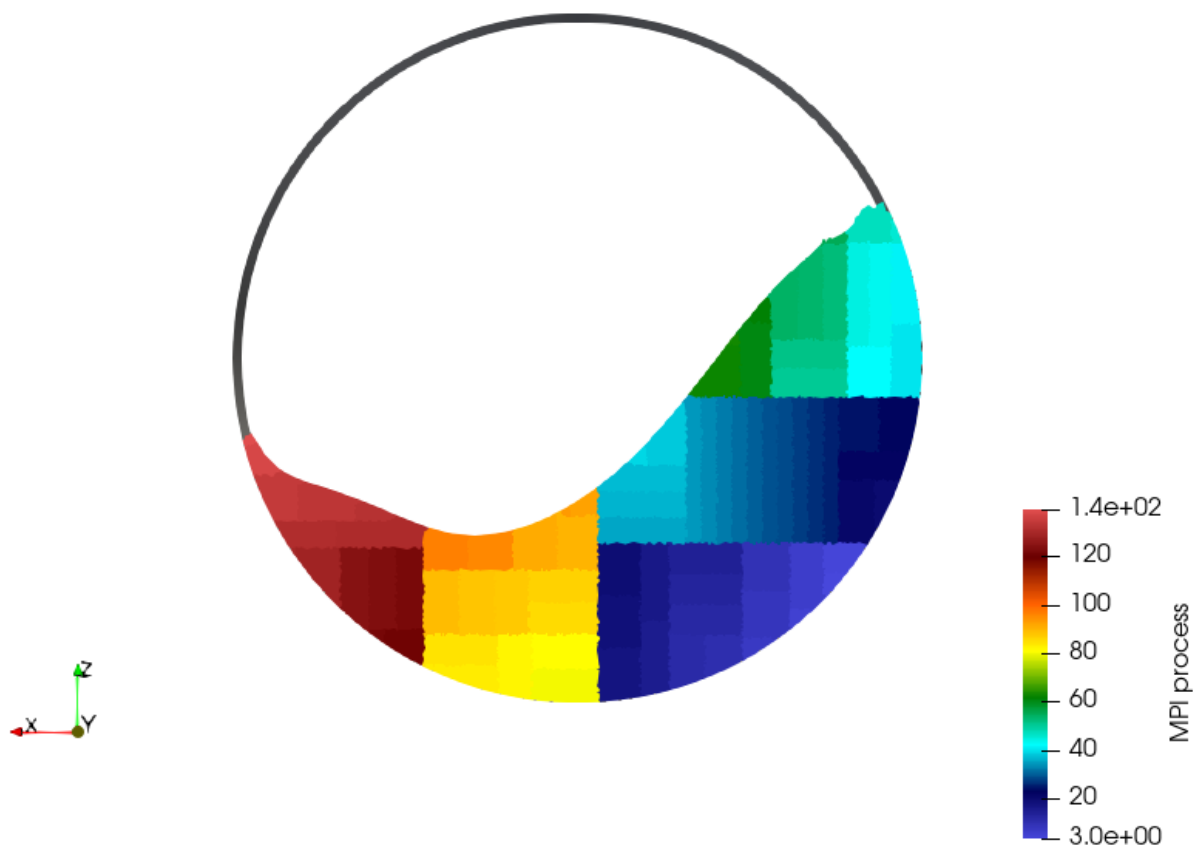


Fig. 2: Domain decomposition of 100,000 spheres in a rotating drum

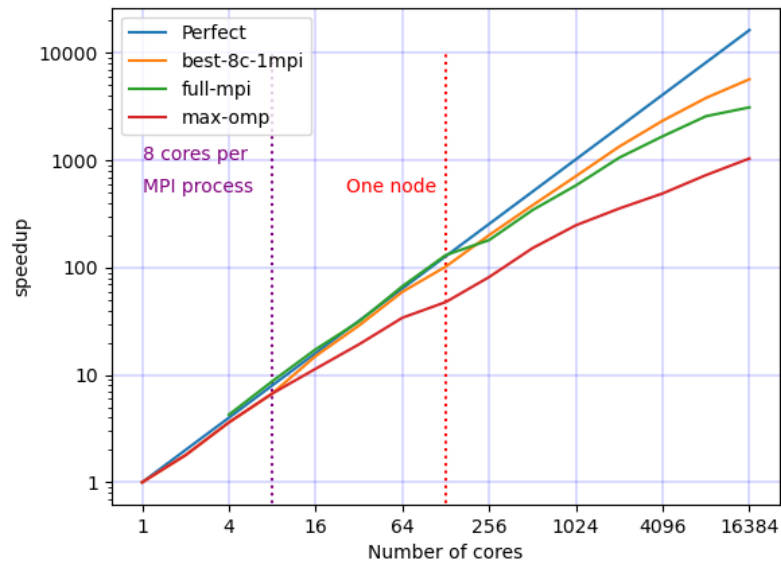


Fig. 3: Speedup for different OpenMP/MPI configurations. ExaDEM simulation with 1, 8, and 128 threads per `mpi` process.

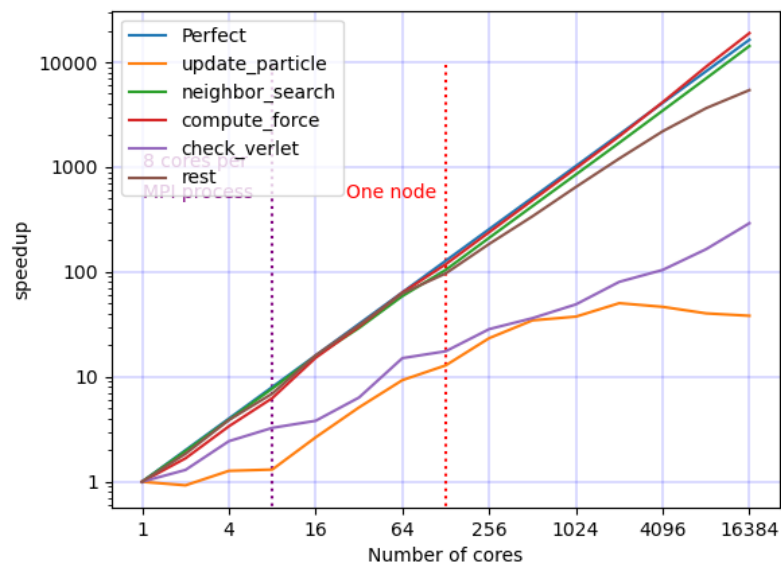


Fig. 4: Operator speedup according to the total number of cores used.

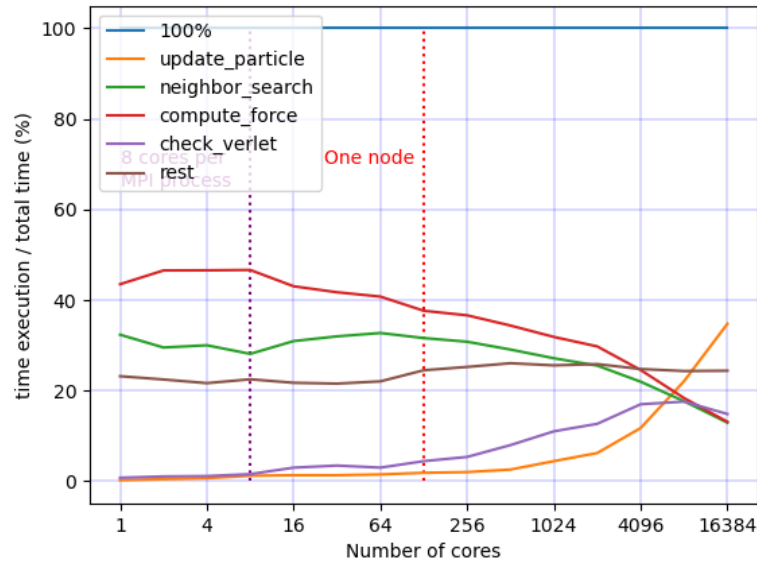


Fig. 5: Operator time ratios at different parallelization scales.

5.3 Contributing Guide

Contributions to *exaDEM* are greatly appreciated.

Please take a moment to review this document in order to make the contribution process easy and effective for everyone involved.

Following these guidelines helps to communicate that you respect the time of the developers managing and developing this open source project. In return, they should reciprocate that respect in addressing your issue or assessing patches and features.

5.3.1 Using the issue tracker

The issue tracker is the preferred channel for bug reports, feature requests and submitting pull requests, but please respect the following restrictions:

- Please do not use the issue tracker for personal support requests, open a “discussion” so that others can benefit from the exchanges and answers (or contact the authors directly at raphael.prat@cea.fr for very specific personal support requests).

5.3.2 Bug reports

A bug is a demonstrable problem that is caused by the code in the repository. Good bug reports are extremely helpful - thank you!

5.3.2.1 Guidelines for bug reports:

- Use the GitHub issue search: check if the issue has already been reported.
- Check if the issue has been fixed: try to reproduce it using the latest master or development branch in the repository.
- Isolate the problem: ideally create a “[reduced test case]”.

- Tag your new issue with the label “bug”.

A good bug report shouldn't leave others needing to chase you up for more information. Please try to be as detailed as possible in your report. What is your environment? What steps will reproduce the issue? What compiler(s) and OS experience the problem? What would you expect to be the outcome? All these details will help people to fix any potential bugs.

5.3.3 Pull requests

Good pull requests - patches, improvements, new features - are a fantastic help. They should remain focused in scope and avoid containing unrelated commits.

Please ask first before embarking on any significant pull request (e.g. implementing features, refactoring code, porting to a different language), otherwise you risk spending a lot of time working on something that the project's developers might not want to merge into the project.

Please adhere to the coding conventions used throughout a project (indentation, accurate comments, etc.) and any other requirements (such as test coverage).

Feature requests are welcome. But take a moment to find out whether your idea fits with the scope and aims of the project. It's up to you to make a strong case to convince the project's developers of the merits of this feature. Please provide as much detail and context as possible.

5.3.3.1 Style

This project is formatted with clang-format using the Google style with a ColumnLimit of 120. Please run clang-format before sending a pull request.

If you use vscode, please add to your settings.json file:

```
{
  "clang-format.style": "{BasedOnStyle: Google, ColumnLimit: 120}",
  "C_Cpp.errorSquiggles": "disabled",
  "files.associations": {
    "*.cu": "cpp",
    "*.hpp": "cpp"
  },
  "[cpp]": {
    "editor.defaultFormatter": "xaver.clang-format",
    "editor.formatOnSave": true
  },
}
```

Commit messages should be in the imperative mood, as described in the Git contributing file:

Describe your changes in imperative mood, e.g. "make xyzzy do frotz" instead of "[This patch] makes xyzzy do frotz" or "[I] changed xyzzy to do frotz", as if you are giving orders to the codebase to change its behaviour.

5.3.3.2 Tests

Please verify the tests pass. Use the following commands in your build directory:

```
ctest --test-dir example
```

If you are adding functionality, add non-regression tests accordingly.

5.3.4 Pull request process

Given the size of the code, only one reviewer will be assigned to you. If the assignment time exceeds one or two weeks, please tag *rprat-pro* or *carrardt* in the conversation.

IMPORTANT: By submitting a patch, you agree to allow the project owners to license your work under the terms of the Apache2.0 License.

5.4 Contributors

5.4.1 Main developers

- Raphaël Prat (CEA/DES) (raphael.prat@cea.fr)
- Thierry Carrard (CEA/DAM)
- Carlo Elia Doncecchi (CEA/DES)

5.4.2 Developers

- Paul Lafourcade (CEA/DAM)
- Lhassan Amarsid (CEA/DES)
- Vincent Richefeu (CNRS)

PYTHON PRE/POST PROCESSING TOOLKIT

The Python script files can be found in the `scripts` folder.

6.1 Overview

The `exaDEM/script` folder provides a collection of Python scripts dedicated to pre-processing and post-processing tasks for exaDEM simulations. Its goal is to streamline the preparation of input data and the analysis of simulation outputs.

The project is organized into two main components:

- a core Python library (`scripts/lib`) gathers all generic functionalities, including data structures (particles, interactions, geometries), input/output utilities, and various computational tools.
- standalone scripts (located in `scripts/pre_processing` and `scripts/post_processing`) implement specific use cases such as configuration generation or result analysis. These scripts rely on the shared library.

Environment: The command (`source /scripts/setenv.sh`) automatically configures the `PYTHONPATH` so that the internal library can be used without requiring installation. Once the environment is sourced, users can run the scripts from any location while still accessing the project's modules.

6.2 Pre-processing

The pre-processing tools are responsible for generating simulation-ready input files from geometric or mesh-based representations.

6.2.1 tess2rockable.py

6.2.1.1 Overview

This script converts a **Neper .tess tessellation file** into a complete **Rockable-compatible input configuration**. It performs:

- parsing of polyhedral cells
- geometric reconstruction of Voronoi-like structures
- Minkowski erosion to account for particle radius
- generation of Rockable shapefile (.shp)
- generation of a full simulation configuration with “sticked particles”

The goal is to transform a geometric tessellation into a physically usable DEM input.

6.2.1.2 Workflow

1. Read Neper tessellation file (*.tess*)
2. Extract vertices, faces, edges, and polyhedra
3. Compute local face normals oriented toward cell centers
4. Perform Minkowski erosion of each cell:
 - each face plane is shifted by a distance *gap*
 - intersections of shifted planes define the eroded polyhedron
5. Compute:
 - volume
 - center of mass
 - inertia tensor (Monte Carlo integration)
6. Write:
 - Rockable shapefile (*.shp*)
 - Rockable configuration file (*_sticked.conf*)

6.2.1.3 Inputs

- `input.tess` (Neper tessellation file)
- `radius` (float) - particle radius used for erosion and stick distance

6.2.1.4 Outputs

- `output.shp`
 - Rockable-compatible geometry file
 - contains vertices, faces, edges, inertia tensor, volume
- `output_sticked.conf`
 - full simulation configuration
 - includes:
 - * particle initialization
 - * interaction laws
 - * gravity and timestep settings
 - * `stickVerticesInClusters` parameter

6.2.1.5 Command line usage

```
python tess2rockable.py input.tess 0.1 output.shp
```

6.2.1.6 Physical assumptions

- Cells are assumed to behave as rigid polyhedra
- Erosion ensures no overlap between particles
- Contact behavior is governed by Rockable's `StickedLinks` law

- Stick distance is set to $4 \times \text{gap}$ (where $\text{gap} = 2 \times \text{radius}$)

6.3 Post-processing

The post-processing tools are used to analyze DEM simulation outputs and extract physical quantities such as stress tensors, strains, and boundary forces.

6.3.1 compute_clusters_stresses_tensors.py

6.3.1.1 Overview

This script computes:

- stress tensors per cluster of particles
- global wall forces
- global deformation (strain) of the particle assembly

It processes time series of:

- *.xyz particle snapshots
- corresponding interaction files

The goal is to extract continuum-scale mechanical quantities from discrete simulations.

6.3.1.2 Workflow

For each simulation timestep:

1. Load particle snapshot (.xyz)
2. Load corresponding interaction file
3. Build particle index and cluster grouping
4. Compute stress tensor:
 - contact forces
 - contact positions
 - normalization by cluster volume (or particle density)
5. Compute wall forces:
 - separation of contacts by spatial position
 - aggregation of forces on top/bottom/left/right boundaries
6. Compute deformation:
 - global bounding box of particles
 - reference configuration stored at first timestep
 - strain computed as relative variation

6.3.1.3 Inputs

- particle files:
 - *.xyz (DEM snapshots)
 - format: position, velocity, cluster id, etc.

- generated in ExaDEM datasets by :

```
analyses:  
- timestep_file: "OutputRockableTestDir/ExaDEMANalyses/dem_pos_vel_%09d.xyz"  
- write_xyz:  
  fields: [ id, velocity, fx, fy, fz, vrot, arot, cluster, mass ]
```

- interaction files:
 - InteractionOutputDir-*/InteractionOutputDir-*_0.txt
 - generated in ExaDEM datasets by :

```
global:  
  symetric: true  
  analysis_interaction_dump_frequency: 500
```

6.3.1.4 Outputs

- clusters_stresses_tensor.txt
Time series of stress tensors per cluster:

```
time cluster_id sxx syy szz sxy sxz syz szx szy
```

- wall_forces.txt
Time series of global mechanical response:

```
time delta_x delta_y delta_z eps_x eps_y eps_z Fx Fy Fz
```

6.3.1.5 Command line usage

```
python compute_clusters_stresses_tensors.py --dir . --density 2500
```

Arguments:

- --dir : directory containing simulation outputs
- --density : particle material density (used for stress normalization)

6.3.1.6 Physical meaning

- Stress tensor:
 - computed from contact force moments
 - normalized by cluster volume (or mass/density approximation)
- Wall forces:
 - represent macroscopic boundary reactions
 - split between top/bottom regions
- Strain:
 - computed from initial bounding box
 - gives global deformation of the granular assembly

6.3.1.7 Limitations

- Assumes correct pairing between .xyz and interaction files

TUTORIALS

The tutorial files can be found in the `example/tutorial` folder.

7.1 Polyhedra tutorials

7.1.1 Build Your Rotating Drum Simulation With Hexapods And A Particle Generator

Warning

The tutorial does not work correctly with the example files in exaDEM version *1.0.2*. Please go to the master or manually change the AREA of the generator.msp to: bounds: [[0 , 0 , 18] , [20 , 7.5 , 20]]

In this section, we will outline the steps to generate a set of hexapods in a rotating drum and then initiate the simulation. The simulation will proceed in three steps:

- The first step involves generating the hexapods within the cylinder. (generator.msp)
- The second step is to wait for all hexapods to be deposited into the cylinder. (wait.msp)
- Finally, the last step is to execute the simulation by setting the cylinder in motion. (run.msp)

The input files are available in the ‘tutorial/hexa-rotating-drum’ folder.

The ingredients for generating the hexapods are:

- defining the polyhedron shape
- setting up the cylinder
- determining the contact force between the polyhedra
- defining a spatial zone for particle generation
- specifying an appearance frequency
- initializing the generated polyhedra

The chosen shape file is a hexapod in the input file, as follows:

```
<
name alpha3
radius 0.237770037899799
preCompDone y
nv 6
```

(continues on next page)

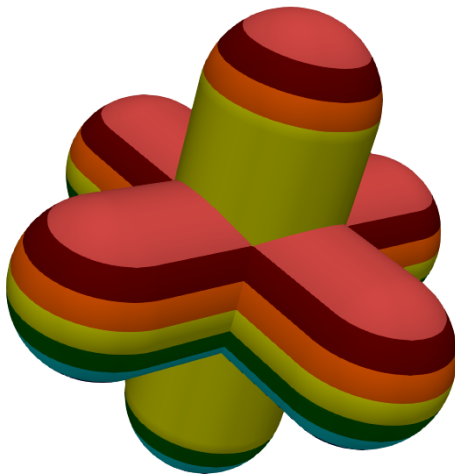
(continued from previous page)

```

0.475540075799599 0 0
-0.475540075799599 0 0
0 0.475540075799599 0
0 -0.475540075799599 0
0 0 0.475540075799599
0 0 -0.475540075799599
ne 3
0 1
2 3
4 5
nf 0
obb.extent 0.713310113699398 0.713310113699398 0.713310113699398
obb.e1 1 0 0
obb.e2 0 1 0
obb.e3 0 0 1
obb.center 0 0 0
volume 0.523598775598299
I/m 0.13266 0.13266 0.13266
>

```

We add a paraview polydata created for this shape (exaDEM doesn't display correctly without face as hexapods, otherwise, during the init step, a *your_shape.vtk* is written).



For this simulation, we choose to fill an infinite cylinder centered at (10,3.75,10) with a radius of 16. This detection is added during the construction of neighbor lists (*nbh_polyhedron* operator).

```

setup_drivers:
- register_cylinder:
  id: 0
  state: {center: [10, 3.75, 10], axis: [1, 0, 1], radius: 16}
  params: { motion_type: STATIONARY }

```

Note

setup_drivers is a default operator integrated in the default execution graph of exaDEM. By default, this operator is set to nop for *no operator*.

We use the contact's law to compute contact force between the polyhedra/polyhedra and cylinder/polyhedra. Gravity is applied everywhere.

```
compute_force:
- gravity_force
- contact_polyhedron:
  symetric: true
  config: { kn: 10000, kt: 10000, kr: 0.0, mu: 0.1, damp_rate: 0.999}
  config_driver: { kn: 10000, kt: 10000, kr: 0.0, mu: 0.1, damp_rate: 0.999}
```

Now, we need to define a spatial zone for particle generation, this zone is the box defined by the inf point = (0,0,19) and sup point (20,7.5,20).

```
particle_regions:
- AREA:
  bounds: [ [ 0 , 0 , 18 ] , [ 20 , 7.5 , 20 ] ]
```

And we design the domain such as the region AREA is included:

```
domain:
  cell_size: 1.5 m
  periodic: [false,true,false]
  grid_dims: [14, 5, 14]
  bounds: [[0 m , 0 m, 0 m], [21 m, 7.5 m, 21 m]]
  expandable: true
```

Now we add a first lattice generator operator to initialize the simulation.

```
add_particles:
- lattice:
  structure: SC
  types: [ 0 ]
  size: [ 1.5 , 1.5 , 1.5 ]
  region: AREA
```

Then we need to initialize hexapods in this region (AREA). The default density is 1, the volume information used to compute the mass is stored in the shape.

```
init_new_particles:
- set_fields:
  polyhedra: true
  type: [ alpha3 ]
  velocity: [ [0,0,-10] ]
  sigma_velocity: [ 0.1 ]
  density: [ 1 ]
  region: AREA
```

Now, we can define our *input_data* operator:

```
input_data:
- read_shape_file:
  filename: alpha3.shp
- add_particles
- init_new_particles
```

The following block consists of the overload of the *add_generated_particles* operator that is set to *nop* by default. Note that this operator is triggered by the frequency *simulation_generator_frequency*: 40000 defined in the global operator, default is -1.

```
add_generated_particles:
- add_particles
- init_new_particles
```

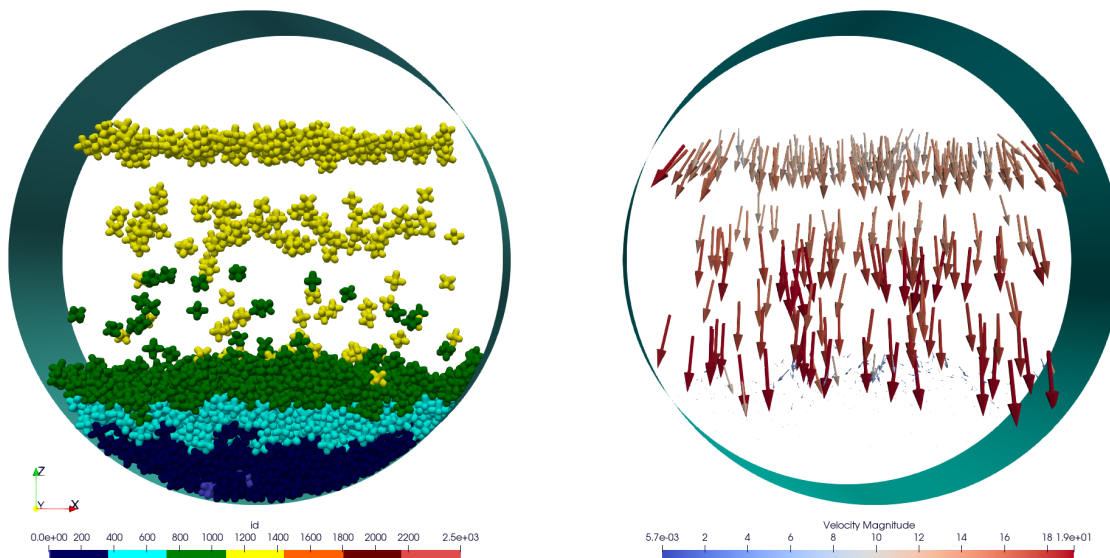
Step one is the *generator.msp* file. To run the simulation, use the following command (around 11 minutes in 12 cores).

```
mpirun -n 2 ./exaDEM generator.msp --omp-num-threads 2
```

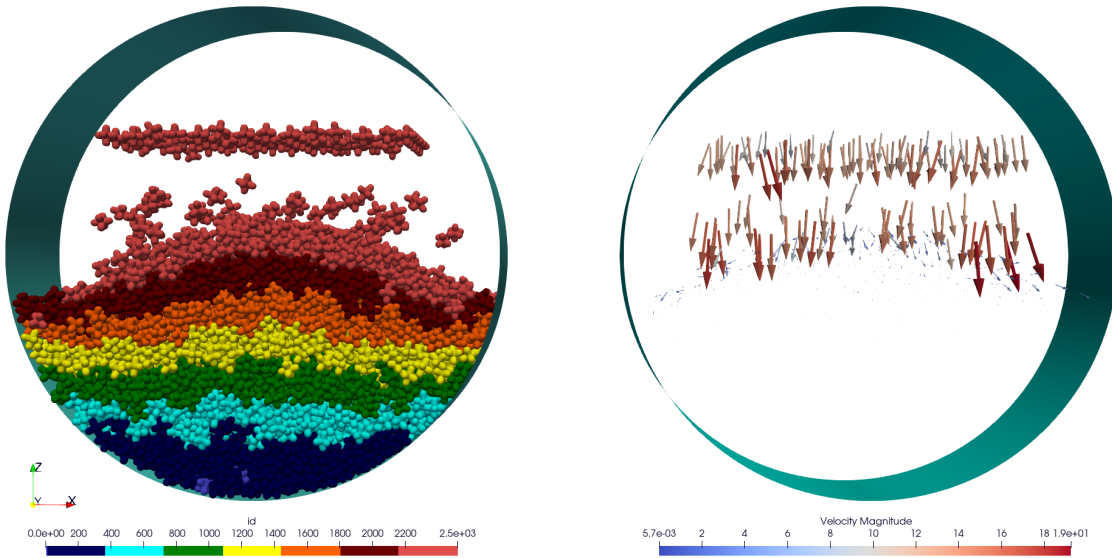
Note

Make sure that the *alpha3.shp* file is in the same location as the simulation.

Picture at the middle of the first step:



Picture at the end of the first step:



Step 2 consists of waiting for the deposit to be finished from timestep 1,200,000 (12s) to 1,400,000 (14s).

First, load the snapshot at time step 1,200,000 and disable generation. It's important not to forget to define the cutoff radius for the hexapods used with the operator *polyhedra_define_radius* for building the Verlet lists.

```
input_data:
- read_shape_file:
  filename: alpha3.shp
- read_dump_particle_interaction:
  filename: ExaDEMOutputDir/CheckpointFiles/exaDEM_0012000000.dump
- radius_from_shape
```

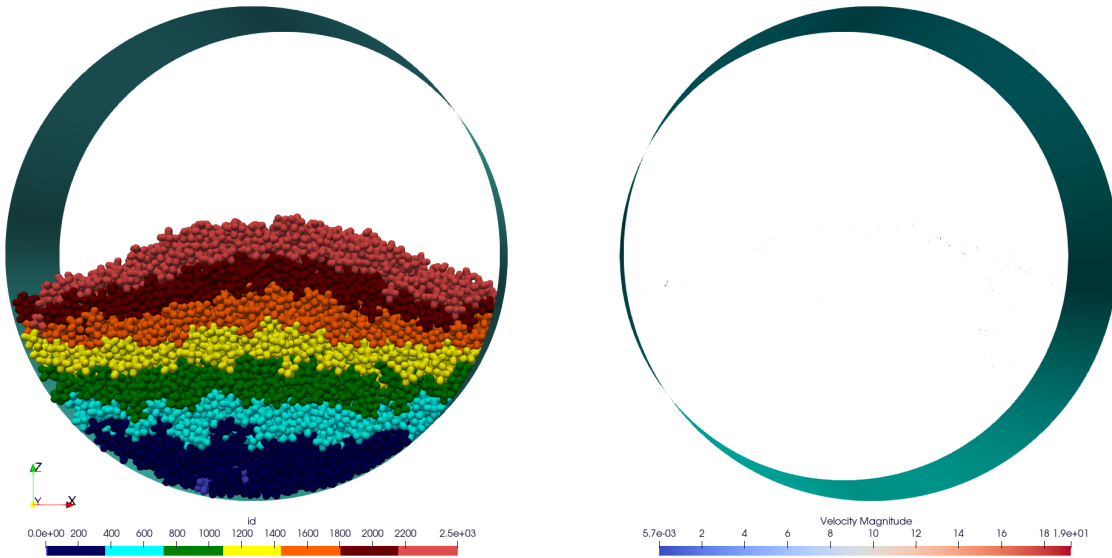
Disable the hexapod generator:

```
simulation_generator_frequency: -1
```

Step two corresponds to the *wait.msp* file. To run this simulation, use the following command (around 4 minutes in 12 cores).

```
mpirun -n 2 ./exaDEM wait.msp --omp-num-threads 2
```

Picture at the end of the second step, the deposit is stable (i.e. no velocity):



Step 3 consists of running the rotating drum simulation from timestep 1,400,000 (14s) to 5,000,000 (50s).

Initiate motion of your drum. You can determine the angular velocity using the Froude number and deduce the angular velocity from it. $Fr = w^2 * R / g$ or $w = \sqrt{Fr * g / R}$. In our case, we desire a cascading behavior with a Froude number of 0.2, $w = \sqrt{0.2 * 9.81 / 16} = 0.350178526 \approx 0.35 \text{ rad.s}^{-1}$.

```
vrot: [0,0.35,0]
```

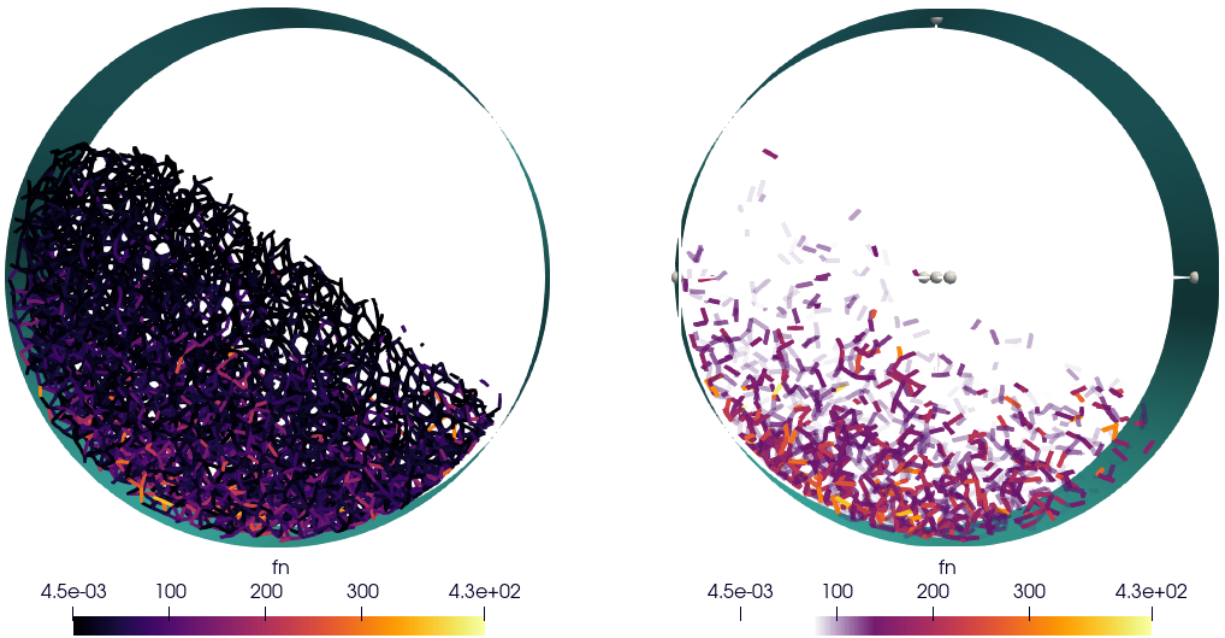
In addition, we display the contact network (normal force) between the hexapods.

```
global:
  enable_contact_network: true
```

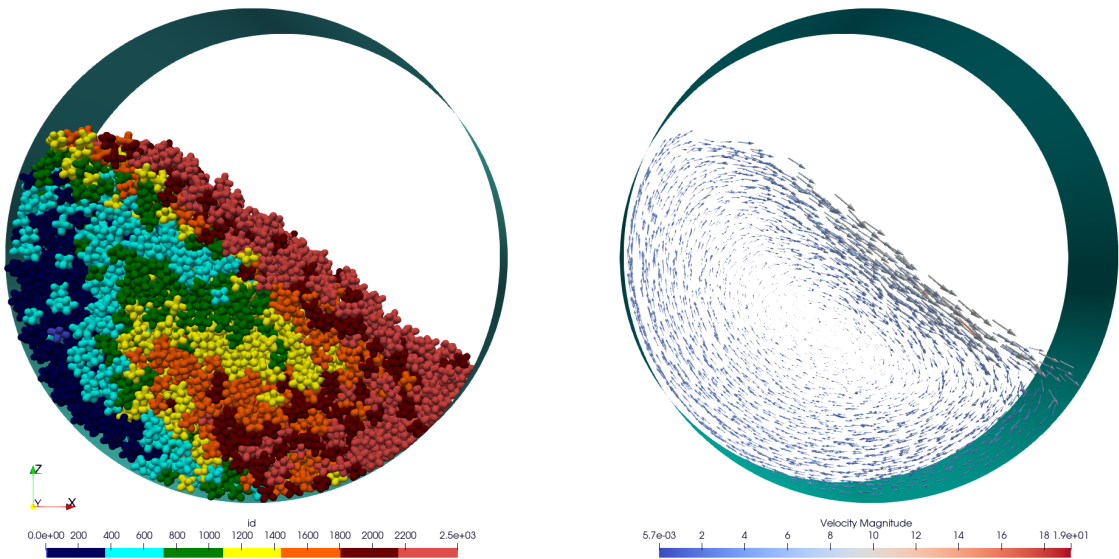
Step three corresponds to the *run.msp* file. To run this simulation, use the following command.

```
mpirun -n 2 ./exaDEM run.msp --omp-num-threads 2
```

This is the final contact network at 50s.



Picture at the end of the third step:



7.1.2 Tutorial: Blade simulation

The aim of this tutorial is to set up a simulation of a descending blade in a silo.

As in the previous example, this simulation is carried out in 3 stages, corresponding to 3 msp files:

- Particle generation [generator.msp]
- Waiting for the deposit to stabilize [wait.msp]

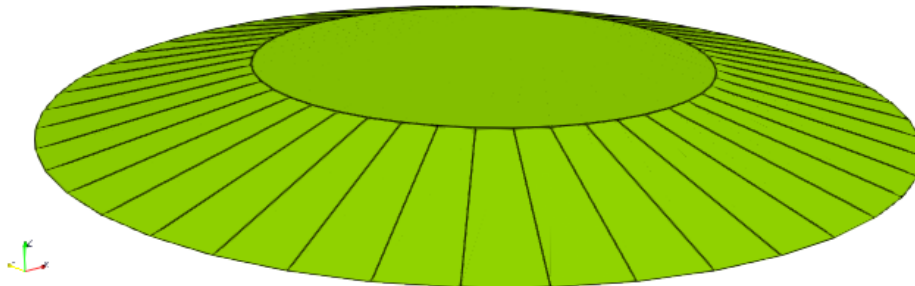
- Adding the blade and setting it in motion [run.msp]

The files are available in the exaDEM/tutorial/blade folder. The STL files are available in the following git: <https://github.com/Collab4exaNBody/exaDEM-Data.git>. Note that msp files are set to fetch STL/SHP files directly from the exaDEM-Data folder if it has been copied to your blade repository.

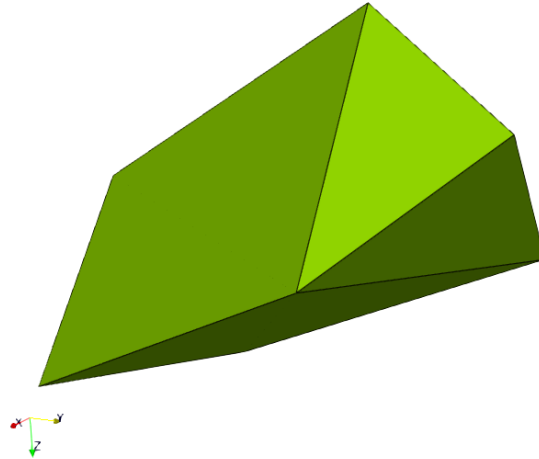
Step 1 consists of generating particles in a cylinder whose main axis is Oz and with a base to stop it. To do this, add them to the list of drivers by defining the *setup_drivers* operator.

```
setup_drivers:  
- register_stl_mesh:  
  id: 0  
  filename: exaDEM-Data/stl_files/mod_base.shp  
  state: {center: [0,0,-20]}  
  params: { motion_type: STATIONARY }  
  minskowski: 0.01  
- register_cylinder:  
  id: 1  
  state: {radius: 25, center: [0,0,0], axis: [1,1,0]}  
  params: { motion_type: STATIONARY }
```

With mod_base, a large shape in the image is just below:



The type of polyhedral particle used for this simulation is as follows (defined into shape.shp):



To add it, define it in the *input_data* operator. We also add particles using the *lattice* operator:

```
input_data:
- read_shape_file:
  filename: shape.shp
- init_rcb_grid
- lattice:
  structure: SC
  types: [ 0 ]
  size: [ 4.0 m , 4.0 m , 4.0 m ]
  region: CYL1 and BOX
- init_fields:
```

In simulations with exaDEM, you need to define a simulation domain, which can be expanded later if necessary, if you specify `expandable: true` and the boundary condition is not periodic. It is very important that cell size, grid dimensions, and simulation box are consistent.

```
domain:
  cell_size: 5.0 m
  periodic: [false,false,false]
  grid_dims: [10, 10, 8]
  bounds: [[-25 m , -25 m , 0 m], [25 m, 25 m, 40 m]]
  expandable: true
```

For this example, we have decided to define a zone/area that fits the shape of the cylinder to generate the particles. To do this, we need to define the different regions:

```
particle_regions:
- CYL1:
  quadric:
    shape: cylz
    transform:
      - scale: [ 23 m, 23 m, 5 m]
      - translate: [ 0 m , 0 m, 50 m ]
- BOX:
```

(continues on next page)

(continued from previous page)

```
bounds: [ [ -25 , -25 , 35 m ] , [ 25 m , 25 m , 40 m ] ]
```

To generate new polyhedra every 25,000 time steps, you need to define two points: the generation frequency and the operator that will create and initialize the particle. The frequency must be specified in the global operator:

```
global:
  simulation_generator_frequency: 25000
```

For the particle generation, operators must be grouped together in the *add_generated_particles* operator call:

```
init_fields:
- radius_from_shape
- set_density:
  density: 0.0026
  region: CYL1 and BOX
- set_rand_velocity:
  var: 0.0001
  mean: [0.0,0.0,-0.5]
  region: CYL1 and BOX
- set_rand_vrot_arot:
  region: CYL1 and BOX
- set_quaternion:
  random: true
  region: CYL1 and BOX
- update_inertia:
  region: CYL1 and BOX

add_generated_particles:
- lattice:
  structure: SC
  types: [ 0 ]
  size: [ 4.0 m , 4.0 m , 4.0 m ]
  region: CYL1 and BOX
- init_fields
```

Next, define the parameters of the contact law and add gravity for gravity deposition. To achieve this, they must be defined in the *compute_force* operator:

```
compute_force:
- gravity_force:
  gravity: [0,0,-0.00981]
- contact_polyhedron:
  symetric: true
  config: { kn: 1.257, kt: 1.077, kr: 0.0, mu: 0.0, damp_rate: 0.999}
  config_driver: { kn: 12.57, kt: 10.77, kr: 0.0, mu: 0.0, damp_rate: 0.999}
```

Note

During deposition, friction is set to 0. It will be changed when the blade is set in motion in step 3.

Finally, we now define the general parameters of the simulation, i.e. the time step value (dt), the number of time steps, the Verlet radius (rcut_inc), and the frequencies of the stop/restart and Paraview outputs.

```

global:
simulation_dump_frequency: 100000
simulation_end_iteration: 1400000
simulation_log_frequency: 1000
simulation_paraview_frequency: 5000
simulation_load_balance_frequency: -1 #27000
dt: 0.0005 s
rcut_inc: 0.5 m
enable_stl_mesh: true
simulation_generator_frequency: 25000

```

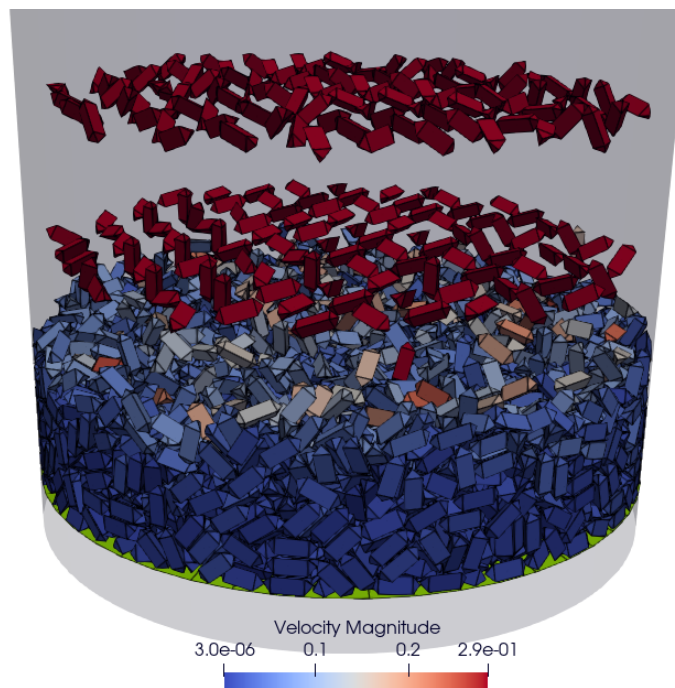
Note

For the sake of performance, it's important to understand that a larger Verlet radius means less frequent updating of interaction lists but more frequent updating of interaction lists. It's very important to find a good trade-off between these two factors.

Step one is the *generator.msp* file. To run the simulation, use the following command.

```
./exaDEM generator.msp --omp-num-threads 12
```

After 1,400,000 time steps, you should reach the following configuration. To complete the deposit, you'll need to wait a while for the deposit to stabilize (step2).



Step 2, in this step, we restart the simulation where we stopped it while disabling the polyhedron generator. To do this, load all stored elements (drivers, shapes, and particles) into the ExaDEMOutputDIR/CheckpointFiles/ folder. For drivers, you can simply include the *.msp* files created for this purpose. For particles, you need to specify in the input_data operator the shape file created and the file containing the particle data and active interactions.

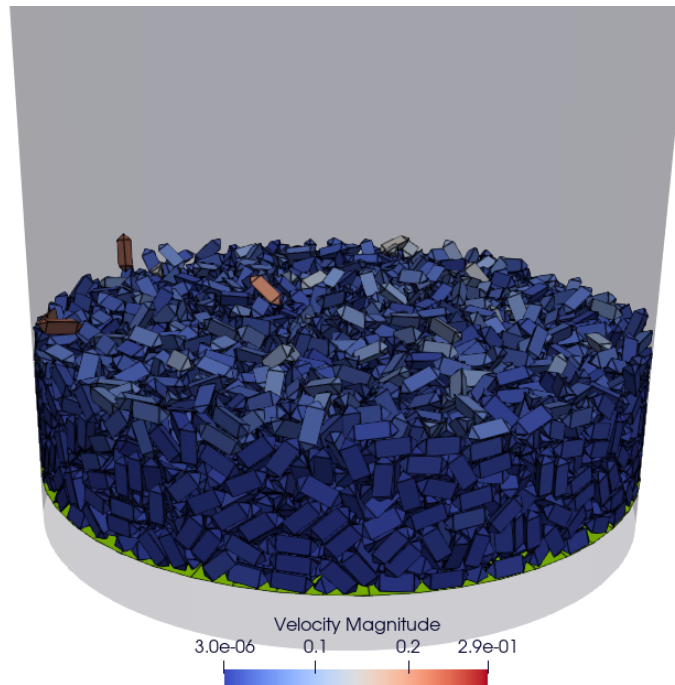
```
includes:
- config_polyhedra.msp
- ExaDEMOutputDir/CheckpointFiles/driver_0001400000.msp

input_data:
- read_shape_file:
  filename: ExaDEMOutputDir/CheckpointFiles/RestartShapeFile.shp
- read_dump_particle_interaction:
  filename: ExaDEMOutputDir/CheckpointFiles/exadem_0001400000.dump
- radius_from_shape
```

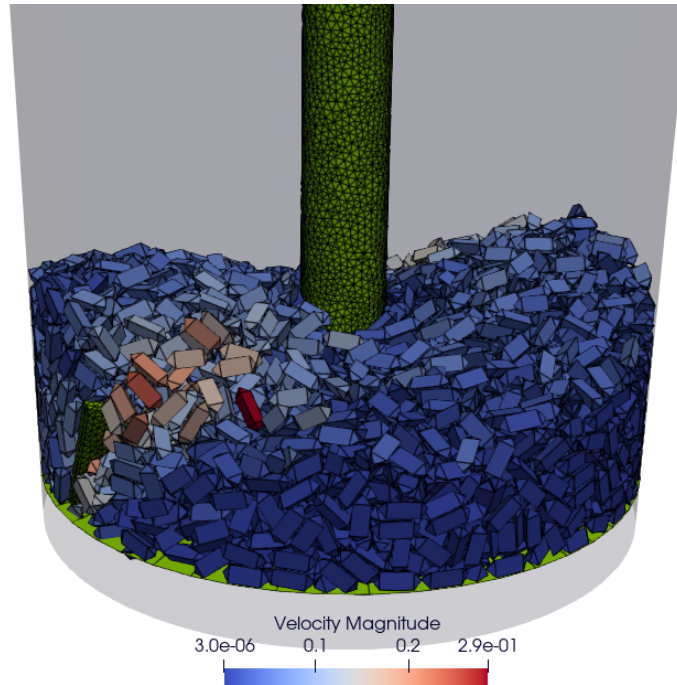
Warning

You have to call `radius_from_shape` otherwise, the interactions won't be defined.

After 100,000 time steps [$t = 1,500,000$ time steps], you should reach the following configuration:



Step 3,



7.2 Developers Tutorials

7.2.1 Add Your Own mutator_field Operator

This is a minimal example to add your own mutator_field operator:

- [1] Set class name: *SetYourFields*
- [2] Set fields: *field::_YOUR_FIELD_1, field::_YOUR_FIELD_2, ..., field::_YOUR_FIELD_N*
- [3] Set types: *YOUR_TYPE_1, YOUR_TYPE_Z, ... , YOUR_TYPE_N*
- [4] Set field slots: *your_field_1, your_field_2, ..., your_field_N*
- [5] Set operator name: *set_your_fields*
- [6] Specify template: *SetYourFields*

```
#include <exaDEM/set_fields.h>
namespace exaDEM
{
    using namespace exanb;
    template<typename GridT
        , class = AssertGridHasFields< GridT, field::_YOUR_FIELD_1, field::_YOUR_FIELD_2, ..
→ .., field::_YOUR_FIELD_N>
        >
    class SetYourFields : public OperatorNode
    {
        static constexpr YOUR_TYPE_1 default_field_value_1 = YOUR_TYPE_1();
        static constexpr YOUR_TYPE_2 default_field_value_2 = YOUR_TYPE_2();
        ...
        static constexpr YOUR_TYPE_N default_field_value_N = YOUR_TYPE_N();
        using ComputeFields = FieldSet< field::_YOUR_FIELD_1, field::_YOUR_FIELD_2, ...,
```

(continues on next page)

```

↪field::_YOUR_FIELD_N>;
    static constexpr ComputeFields compute_field_set {};

    ADD_SLOT( GridT, grid , INPUT_OUTPUT );
    ADD_SLOT( YOUR_TYPE_1, your_field_1, INPUT, default_radius, DocString{"default ↪
↪value for all particles"} );
    ADD_SLOT( YOUR_TYPE_2, your_field_2, INPUT, default_radius, DocString{"default ↪
↪value for all particles"} );
    ...
    ADD_SLOT( YOUR_TYPE_N, your_field_N, INPUT, default_radius, DocString{"default ↪
↪value for all particles"} );

public:

    inline std::string documentation() const override final
    {
        return R"EOF(
            This operator sets the ... value(s) for every particles.
        )EOF";
    }

    inline void execute () override final
    {
        SetFunctor<YOUR_TYPE_1,YOUR_TYPE_2, ... , YOUR_TYPE_N> func = {
            { *your_field_1 },
            { *your_field_2 },
            ... ,
            { *your_field_N }
        };
        compute_cell_particles(
            *grid , false , func ,
            compute_field_set ,
            gpu_execution_context() ,
            gpu_time_account_func()
        );
    }
};
template<class GridT> using SetYourFieldsTmpl = SetYourFields<GridT>;
// === register factories ===
CONSTRUCTOR_FUNCTION
{
    OperatorNodeFactory::instance()->register_factory( "set_your_fields", make_grid_
↪variant_operator< SetYourFieldsTmpl > );
}
}

```


PUBLIC EXAMPLES

This section lists a set of public examples illustrating some of the possibilities of `exaDEM`, but which are not necessarily intended to be maintained.

8.1 Example 1

Repository: <https://github.com/Collab4exaNBody/exadem-public-examples/tree/main/ex1>

- Version tested: v1.1.6
- Particle type: cylinder
- Driver type: Cylinder
- Motion type: STATIONARY with Rotation

8.2 Example 2

Repository: <https://github.com/Collab4exaNBody/exadem-public-examples/tree/main/ex2>

- Version tested: v1.1.6
- Particle type: Toupie
- Driver type: RShape
- Motion type: STATIONARY

8.3 Example 3

Repository: <https://github.com/Collab4exaNBody/exadem-public-examples/tree/main/ex3>

- Version tested: v1.2.0
- Particle type: Cube
- Driver type: RShape
- Motion type: PARTICLE

EXANBODY: FRAMEWORK FOR N-BODY SIMULATIONS ON HPC PLATFORMS

This framework, developed at the French Atomic Agency (CEA), is tailored for N-Body simulations on High-Performance Computing (HPC) platforms. Originally designed for the ExaSTAMP Molecular Dynamics code, it has been extended to cater to various N-Body problems.

Key Characteristics:

- **Language:** Implemented in C++17, leveraging modern language features for efficiency and versatility.
- **Parallelization:: Hybrid approach integrating:**
 - Vectorization for CPU optimization.
 - Thread-parallelization using OpenMP for multi-core architectures.
 - GPU-parallelization via CUDA to harness GPU computational power.
 - MPI-parallelization for distributed memory systems.
- **Spatial Domain Decomposition:** Utilizes spatial domain decomposition techniques for efficient workload distribution among processors.
- **Load Balancing (RCB):** Implements Load Balancing using Recursive Coordinate Bisection (RCB) for optimal task distribution among processing units.
- **Parallel IO:** Enables efficient handling of parallel Input/Output operations. Supports checkpoint files, parallel Paraview files, and diagnostics in a parallelized manner.
- **In-situ Analysis:** Provides real-time data analysis capabilities during simulation execution, minimizing data movement and storage overhead.

9.1 ExaDEM Variant

ExaDEM is a software solution in the field of computational simulations. It's a Discrete Element Method (DEM) code developed within the `exaNBody` framework. This framework provides the basis for DEM functionalities and performance optimizations. A notable aspect of ExaDEM is its hybrid parallelization approach, which combines the use of MPI (Message Passing Interface) and Threads (OpenMP). This combination aims to enhance computation times for simulations, making them more efficient and manageable.

Additionally, ExaDEM offers compatibility with MPI and GPUs, using the CUDA programming model (Onika layer). This feature provides the option to leverage GPU processing power for potential performance gains in simulations. Written in C++17, ExaDEM is built on a contemporary codebase. It aims to provide researchers and engineers with a tool for addressing DEM simulations.

9.2 ExaSPH Variant

9.3 ExaSTAMP Variant

INSTALLING EXANB

To proceed with the installation, your system must meet the minimum prerequisites. The first step involves the installation of exaNBody:

```
git clone https://github.com/Collab4exaNBody/exaNBody.git
mkdir build-exaNBody/ && cd build-exaNBody/
cmake ../exaNBody/ -DCMAKE_INSTALL_PREFIX=path_to_install
make install
export exaNBody_DIR=path_to_install
```


N-BODY BACKGROUND

11.1 N-body methods

N-body methods encompass a variety of techniques used to model the behavior and interactions of a set of particles over time. These methods consist in solving Newton's equation of motion $f = ma$ for each particle at each time step, where f corresponds to the sum of the forces applied to the particle, a its acceleration and m its mass. The forces are deduced from the interactions between particles according to their types, i.e. contact, short-range, or long-range interactions, and external forces applied to the sample (i.e. gravity). Velocities are then deduced from the accelerations and subsequently used to update the particle positions at the next time step. This process is repeated, typically with a fixed time step t , according to an integration scheme⁴ until the desired duration is reached. The collection of particle configurations over time allows to study a wide range of phenomena, from granular media movements, with the Discrete Element Method (Dem), to material crystal plasticity at the atomic scale using Molecular Dynamics (Md), going up to the galaxy formation with the Smoothed-Particle Hydrodynamics (Sph).

11.2 N-body simulation codes

The development of a N-body code is led by the need to figure out the neighborhood of a given particle for every timestep in order to process interactions of different kinds.

Particle interactions can be categorized as short-range and long-range. Short-range interactions are considered negligible beyond a specified cut-off radius. To optimize calculations, neighboring particle detection algorithms are employed to eliminate unnecessary computations with distant particles. Each N-Body method employs a wide variety of short-range interactions that capture different particle physics. For example, visco-elastic contacts in Discret Element Method follow Hooke's law or Hertz law to model contact elasticity between rigid particles, while pair potentials like Lennard-Jones or Morse are used for gas or liquid atom interactions in classical Md. Long-range interactions, on the other hand, can sometimes not be neglected and result in algorithmic complexity of $O(N^2)$. Such interactions, like gravitation in astrophysics, or electrostatic forces in Molecular Dynamics, are typically modeled using the Ewald summation method. Fortunately, calculation approaches such as the fast multipole method can achieve a complexity of $O(N)$, thanks to an octree structure, and can be efficiently parallelized. Although this paper primarily focuses on short-range interactions, both types of interaction can be dealt with in exaNBody.

Neighbor lists are built, using different strategies, to shorten the process of finding out the neighbors of a particle within the simulation domain. It helps optimizing the default algorithm having a complexity of $O(N^2)$ that tests every pair of particles (if N is the number of particles). The most common strategy to deal with any kind of simulation (static or dynamic, homogeneous and heterogeneous density) is a fusion between the linked-cell method, see figure 2, and the Verlet list, see figure 2 method. The combination of these methods has a complexity of $O(N)$ and a refresh rate that depends on the displacement of the fastest particle. This algorithm is easily thread-parallelized. Others less-used neighbor search strategies have been developed to address specific simulations

Domain decomposition is usually employed in N-Body methods to address distributed memory parallelization, assigning one subdomain to each MPI process. This implies the addition of ghost areas (replicated particles) around subdomains to ensure each particle has access to its neighborhood. Over time, many algorithms have been designed

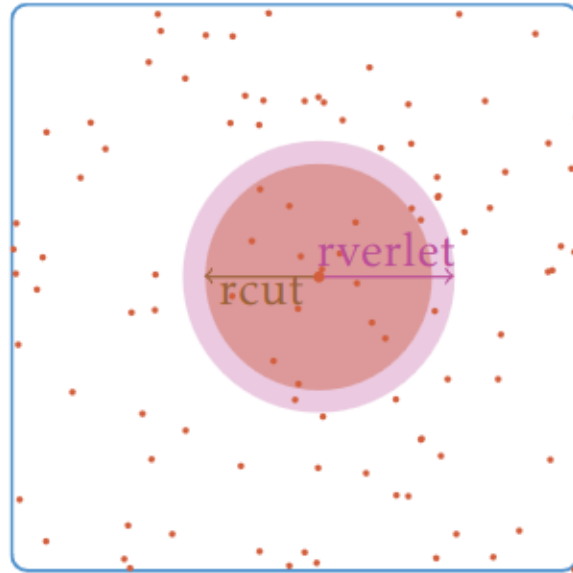


Fig. 1: Figure 1: A method for building neighbor lists using the Verlet lists. r_{cut} is the radius cut-off and r_{Verlet} is the radius of Verlet.

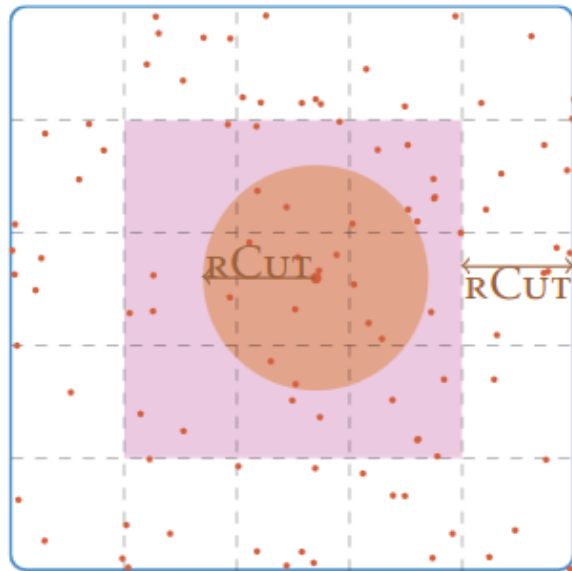


Fig. 2: Figure 2: A method for building neighbor lists using linked cells. r_{cut} is the radius cut-off.

to improve load-balancing such as: Recursive Coordinate Bisection (RCB), the Recursive Inertial Bisection (RIB), the Space Filling Curve (SFC), or graph method with ParMETIS. Note that the library Zoltan gathers the most popular methods. To ease neighbor list construction (i.e. employing the linked-cell method), the simulation domain is described as a cartesian grid of cells, each of which containing embedded particles. Each subdomain then consists in a grid of entire cells assigned to one Mpi process. In contrast, concurrent iteration over the cells of one subdomain's grid provides the basis for thread parallelization at the NUMA node level.

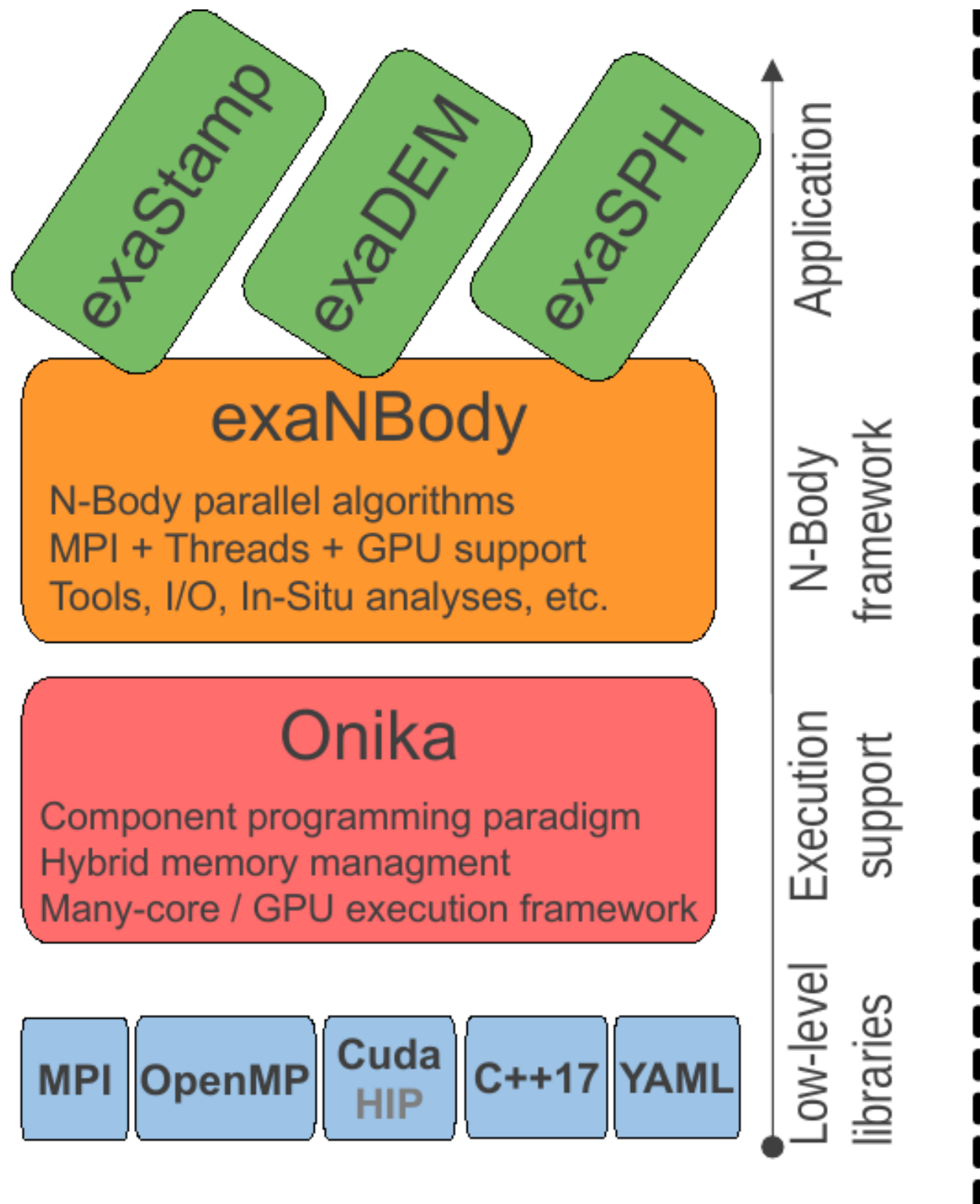
Commonalities are shared across N-body simulation codes, such as numerical schemes, neighbor particle detection, or short/long-range interactions. The computation time dedicated to interaction and force calculations can be significant (over 80% of the total time) depending on the studied phenomenon complexity. Additional factors, such as neighbor lists construction computational cost, can impact overall simulation time. For dynamic simulations involving rapidly moving particles and computationally inexpensive interactions, more than 50% of the total time may be spent on neighbor list construction. The computationally intensive sections of the code vary depending on methods and phenomena studied, requiring optimizations such as:

- MPI parallelization
- Vectorization
- Multi-threading
- GPU usage

SOFTWARE STACK OF EXANBODY

12.1 Software Stack Layers:

1. **External Libraries:** This layer encompasses external dependencies utilized by the software. These libraries are leveraged to access functionalities that are not internally developed, enhancing the overall capabilities of the system.
 - **HPC Libraries:** High-Performance Computing (HPC) libraries serve as powerful tools within the software stack, providing optimized routines and functionalities specifically designed for high-computational tasks.
2. **Low-Level and Runtime Support (Onika):** Foundational infrastructure and runtime support are provided by this layer. Onika serves as the backbone, offering essential utilities for the execution and management of the software. It's important to note that ongoing efforts are directed towards integrating frameworks like OpenMP or CUDA.
3. **N-Body Framework:** Positioned at the core of the software stack, the N-Body Framework proposes a large variety of N-Body features to build efficiently N-Body simulation such as traditionnal integration schemes, IO, or partiionning methods.
4. **Application Layer:** This layer encapsulates the specific applications built upon the software stack. It provides the interface for users to interact with the functionalities offered by the stack and serves as the platform for N-Body application-specific implementations and functionalities.



12.2 Guidelines for operators:

- **Common Operators:** These operators serve as N-Body fundamental operations utilized across various components of the software, facilitating consistent functionality and reducing redundancy.
- **Application-Specific Operators:** Tailored operators designed to cater to the unique requirements of specific applications built on top of the software stack.
- **Mutualization:** This process involves the consolidation and centralization of specific operators shared by multiple applications. It aims to optimize resources and streamline the execution of these shared functionalities.
- **Specific Operators Shared Among Applications:** For increased efficiency and ease of access, specific operators that are shared among multiple applications are centralized within exaNBody. This consolidation not only ensures optimized utilization but also simplifies management and maintenance, benefiting all associated applications.

PERFORMANCE AND PORTABILITY

The complex and ever-changing architectures of modern supercomputers make it difficult to maintain software performance. exaNBody aims at providing performance portability and sustainability on those supercomputers with robust domain decomposition, automated inter-process communications algorithms, adaptable particle data layout, and a set of hybrid (CPU/GPU) parallelization templates specialized for N-Body problems.

13.1 Application level specialization

First of all, the internal units to be used are specified as well as the physical quantities to be stored as particle attributes. These quantities (or fields), are defined using a symbolic name associated with a type, e.g. velocity as a 3D vector. A field set is a collection of declared fields. One of the available field sets is selected and used at runtime, depending on simulation specific needs. As depicted in figure below, particles are dispatched in cells of a cartesian grid spanning the simulation domain. In short, the data structure containing all particles' data will be shaped as a cartesian grid of cells, each cell containing all fields for all particles it (geometrically) contains. More specifically, the reason why fields and field sets are defined at compile time is that particle data storage at the cell level is handled via a specific structure guaranteeing access performance and low memory footprint.

13.2 Spatial domain decomposition and inter-process communications

The coarsest parallelization level can become the main bottleneck due to network latencies and load imbalance issues. To take advantage of this first level of parallelization, the simulation domain is divided into subdomains using an recursive coordinate bisection (RCB) algorithm, as depicted in figure below, assigning one subdomain to each MPI process.

This is achieved thanks to three main components: cell cost estimator, RCB domain decomposition, and particle migration. Particle migration can be used as-is by any N-Body application, thanks to the underlying generic particle data storage. It supports heavily multi-threaded, large scale, simulations while lowering peak memory usage. Additionally, the migration algorithm is also customizable to fit specific application needs, keeping unchanged the core implementation. For instance, molecular dynamics simulations may transport per-cell data fields and discret element method simulations may migrate friction information related to pair of particles. Finally, ghost particle updates are available to any N-Body application, via customizable components.

13.3 Intra-node parallelization API

This API is available in exaNBody to help developers express parallel computations within a MPI process. This API offers a set of parallelization templates associated with three types of computation kernels:

- Local calculations on a particle (such as resetting force operator)

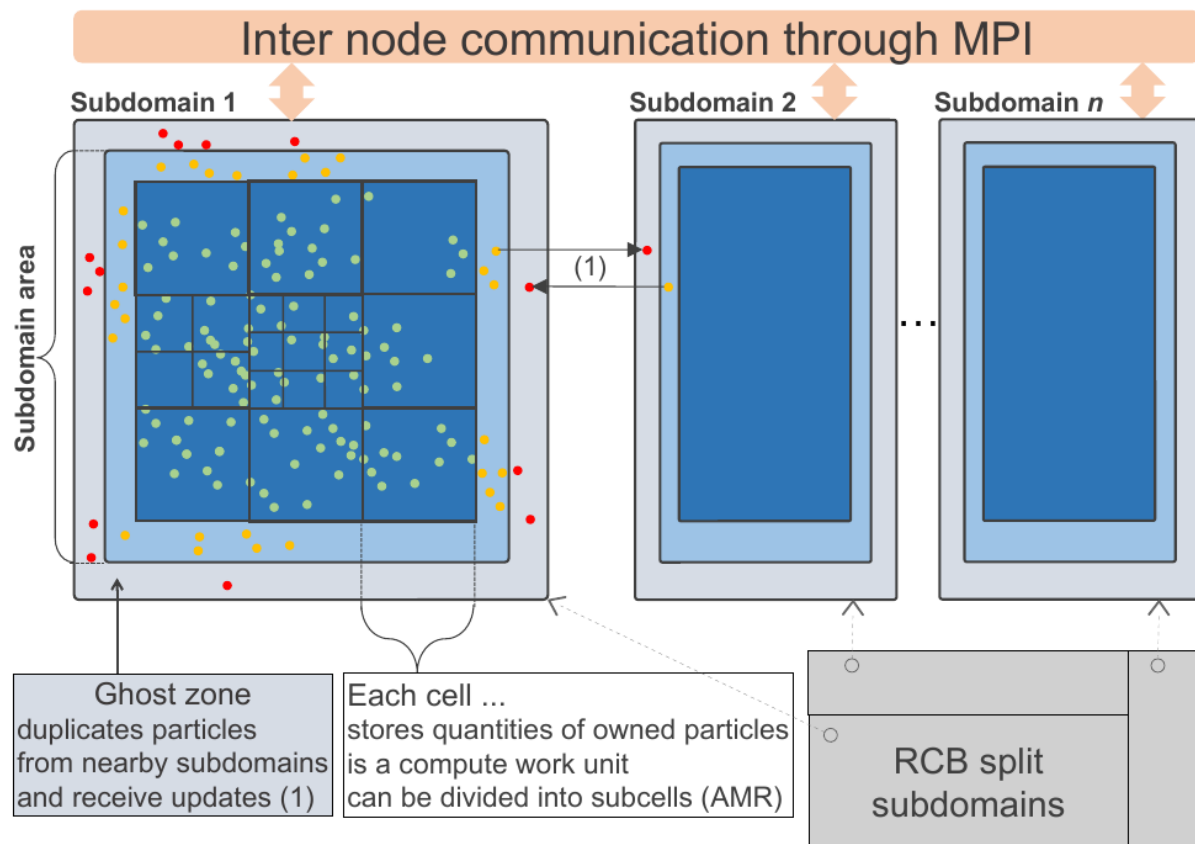


Fig. 1: Figure 1: Overview of domain decomposition and inter process communications in exaNBody framework.

- calculations coupled with reduction across all particles (such as getting the total number of particles or the current temperature)
- calculations involving each particle and its neighbors (such as computing potential forces in molecular dynamics or contact forces in discret element method)

When a developer injects a compute function into these templates, computation may be routed to CPU or GPU, as illustrated in figure below.

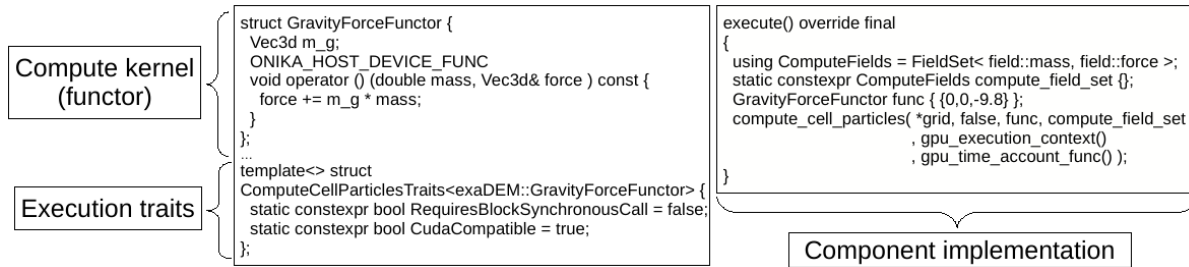


Fig. 2: Figure 2: Example of a particle centered computation executable on both COU and GPU. Three ingredients: a user functor (the kernel), static execution properties (via traits specialization), a ready to use parallelization function template.

While thread parallelization on the CPU is powered by OpenMP, Cuda is employed to execute the computation kernel on the Gpu, using the same provided function. The main difference between the two execution modes is that each cell is a unitary work unit for a single thread in OpenMP context but it is processed by a block of threads in Cuda. Those two parallelization levels (multi-core and GPU) are easily accessible to developers thanks to the execution support layer of Onika. Onika is the low-level software interface that powers exaNBody building blocks. It is responsible for aforementioned data containers, memory management (unified with GPU), and it is the foundation for hybrid execution abstraction layer.

13.4 Particle data layout and auxiliary data structures

These two points are two essential features to maximize performance at the NUMA node level. In exaNBody, particle data are packed independently in each cell using a container specifically designed to match both CPU's SIMD and GPU's thread blocks requirements concerning data alignment and vectorization friendly padding. This generic container, available in ONIKA toolbox, not only adapts to specific hardware characteristics at compile time, but ensures minimal memory footprint with as low as 16 bytes overhead per cell regardless of the number of data fields, allowing for very large and sparse simulation domains. N-Body simulations also heavily depend on particles' neighbors search algorithm and storage structure. The search usually leverages the grid of cell structure to speed up the process, and neighbors lists data structure holds information during several iterations. However, depending on the simulation, particles may move rapidly while their distribution may be heterogeneously dense. Those two factors respectively impact neighbor list update frequency and its memory footprint. On the one hand, exaNBody takes advantage of an Adaptive Mesh Refinement (AMR) grid to accelerate (frequent) neighbor list reconstructions. On the other hand, a compressed neighbor list data structure saves up to 80% of memory (compared to uncompressed lists) while still ensuring fast access from both the CPU and the GPU.

13.5 Flexible and user friendly construction of N-Body simulations

A crucial aspect for software sustainability is to maintain performance over time while managing software complexity and evolution. Complex and rapidly evolving scientific software often encounter common pitfalls, such as code duplication, uncontrolled increase in software inter-dependencies, and obscure data/control flows. This observation has led us to develop our component-based model to avoid these pitfalls. In our model, individual software components are implemented using C++17 and are application structure oblivious, meaning they only know their input and output

data flows. Application obliviousness is a crucial aspect of the present design, promoting reusability while preventing uncontrolled growth of internal software dependencies. Each component is developed as a class, inheriting from a base class `OperatorNode` and explicitly declares its input and output slots (data flow connection points). Once compiled, these components are hierarchically assembled at runtime using a Sequential Task Flow (STF), with a YAML syntax, as shown in figure below.

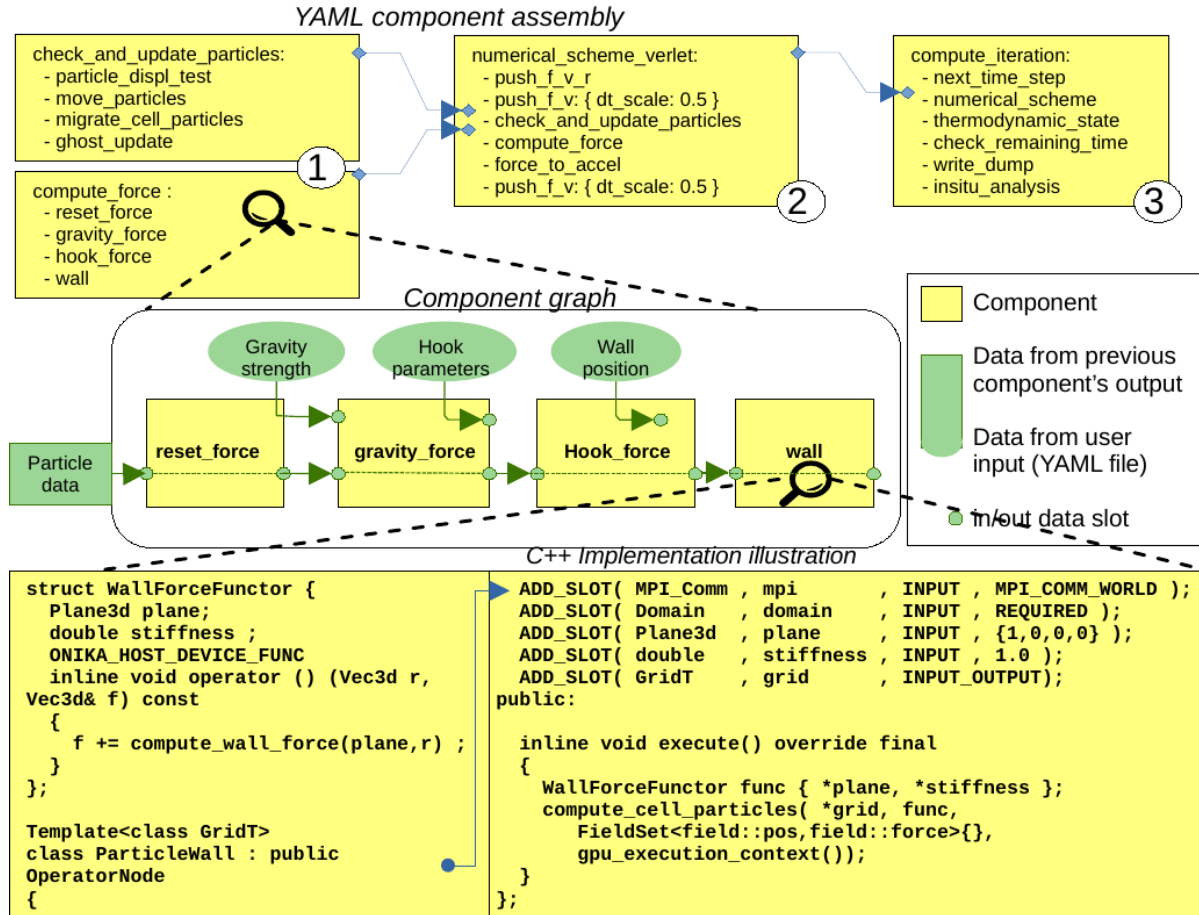


Fig. 3: Figure 3: Illustrative sample of components assembly using YAML description. 1) C++ developed components are assembled and connected in the manner of a STF, creating a batch component. 2) and 3) illustrate batch components aggregation to higher and higher level components, up to full simulation task flow.

A set of base components are already available to developers, embedded within `exaNBody`, such as: common computations, checkpoint/restart, visualization and In-Situ analytics, allowing developers to focus on their application specific components. We also observed that this component based approach not only prevents some development pitfalls, but enables various simulation code structures. YAML formatted component configuration makes it simple for a user to amend or fine tune the simulation process. For instance it can be used to change the numerical scheme or even to insert In-Situ analysis components at specific stages of the simulation process, leveraging In-Situ processing to limit disk I/O. Finally, this component based splitting of the code gives `exaNBody` the opportunity to provide integrated profiling features that automatically give meaningful performance metrics for each part of the simulation. It allows the user to access computation time spent on CPU and GPU, as well as imbalance indicator. It can also interoperate with `nSight` System from NVIDIA and summarize memory footprint with detailed consumption.

ALGORITHM PATTERNS

This is Developer Documentation.

The aim of this section is to explain to developers the tools used in exaNBody to propose default parallel algorithms. For this purpose, we will describe the 4 main algorithmic patterns:

- *block_parallel_for*: exposes a simple parallelization scheme (like a parallel for in OpenMP), but with a notion of “thread blocks” and the possibility of asynchronous execution. It is defined directly in the Onika layer, as it is not specific to particle systems.
- *compute_cell_particles*: performs a parallel processing on all particles contained within a grid (Grid type pattern).
- *reduce_cell_particles*: performs an operation on each particle returning a value, the values obtained being reduced to a single one (e.g., a sum).
- *compute_cell_particle_pairs*: the most important and most used pattern, allows calculations on particles and their neighbors.

Guidelines:

- Support hybrid execution, automatically directing calculations either to the *CPU* cores or to the *GPU*.
- Avoid code duplication between components.
- Avoid the complexities of setting up *OpenMP/CUDA* parallelism.
- Allow asynchronous parallel operations (with results retrieved later in the code).
- Automatically adapt to the different *OpenMP* parallelism modes supported by the component system (simple, nested, and task-based).
- Facilitate the implementation of a single computation function for both *CPU* and *GPU*.
- Manage dynamic load balancing among the *SMs* of the *GPUs* themselves.

14.1 Block Parallel For

This construction is the most rudimentary; it is equivalent to an *OpenMP* parallel for directive but with many additional possibilities. We will look at a step-by-step example where a fixed value is added to all elements of an array of arrays. The example on a two-dimensional array demonstrates the use of two-level multi-threaded parallelism we have discussed, which resembles a bit the array of cells, each containing a set of particles typically processed in exaNBody.

14.1.1 Basic usage

Before calling `block_parallel_for`, it is necessary to define the functor (the calculation function) that will be applied in parallel to our data.

```
namespace tutorial
{
    using namespace exanb;
    // a functor = function applied in parallel
    // it is defined by a class (or struct) with the call operator ()
    struct BlockParallelValueAddFunctor
    {
        Array2DReference m_array;
        double m_value_to_add = 0.0; // value to add
        ONIKA_HOST_DEVICE_FUNC      // works on CPU and GPU
        void operator () (size_t i) const // call operator with i in [0;n[
        {
            // a whole block (all its threads) execute
            ↪iteration i
            const size_t cols = m_array.columns();
            ONIKA_CU_BLOCK_SIMD_FOR(size_t, j, 0, cols) // parallelization among the threads
            ↪of the current block
            {
                // for iterations on j in [0;columns[
                m_array[i][j] += m_value_to_add; // each thread executes 0, 1, or multiple
            ↪iterations of j
            }
        }
    };
}
```

- `ONIKA_CU_BLOCK_SIMD_FOR`: Defines a macro that facilitates the use of the second level of parallelism within threads of the same *CUDA* block.
- Usage:
 - GPU :Ensures that the functor's code is executed across all threads within a *CUDA* block with consistent iteration index *i*. When compiled for the *GPU*, it distributes iterations based on `ONIKA_CU_BLOCK_SIZE` and `ONIKA_CU_THREAD_IDX`.
 - CPU: On the host, this construct transforms into a straightforward for loop. However, on the host, the loop includes the *OpenMP* directive `#pragma omp simd for`, ensuring loop vectorization, even with only one *CPU* thread per block, provided *CPU* and compiler support is available.

Next, we need to inform the compiler whether our functor is compatible with *CUDA*. In some situations, our code may have a strong incompatibility (such as calling a non-compatible library) that would prevent it from compiling correctly with the `nvcc` layer. This needs to be known at compile time (not just at runtime) because otherwise, compilation errors would occur.

```
namespace onika
{
    namespace parallel
    {
        template<>
        struct BlockParallelForFunctorTraits<tutorial::BlockParallelValueAddFunctor>
        {
            static constexpr bool CudaCompatible = true; // or false to prevent the code from
            ↪being compiled with CUDA
        };
    };
}
```

(continues on next page)

(continued from previous page)

```

    };
}
}

```

The final step is to launch the parallel operation in the code of our component:

```

namespace tutorial
{
    using namespace exanb;
    class SynchronousBlockParallelSample : public OperatorNode
    {
        ADD_SLOT(Array2D, my_array, INPUT_OUTPUT, REQUIRED);
        ADD_SLOT(double, my_value, INPUT, 1.0);
    public:
        inline void execute() override final
        {
            using onika::parallel::block_parallel_for;
            if( my_array->rows() == 0 || my_array->columns() == 0 )
            {
                my_array->resize( 1024 , 1024 );
            }
            BlockParallelValueAddFunctor value_add_func = { *my_array // reference our data_
↪ array through its pointer and size
                                                    , *my_value // value to add to the_
↪ elements of the array
                                                    };
            // Launching the parallel operation, which can execute on GPU if the execution_
↪ context allows
            block_parallel_for( my_array->rows() // number of iterations,_
↪ parallelize at the first level over rows
                               , value_add_func // the function to call in_
↪ parallel
                               , parallel_execution_context() // returns the parallel execution_
↪ context associated with this component
                               );
        }
    };
}

```

The corresponding complete exemple is in exaNBody source tree and compiled, ready to be tested, and is linked here after : `synchronous_block_parallel.cpp`

14.1.2 Asynchronous parallel execution

In the following example, we are still using the simplest version of `block_parallel_for`, but we want to trigger parallel execution asynchronously (running in the background), and explicitly wait for completion later on. To this end, we capture object returned by `block_parallel_for`, of type `ParallelExecutionWrapper`, to handle its synchronization manually. When this object is not captured in a variable, it is therefor destructed right after termination of `block_parallel_for`, which has the side effect of launching and waiting for the completion of created parallel operation.

```

namespace tutorial
{
    class AsyncBlockParallelSample : public OperatorNode

```

(continues on next page)

(continued from previous page)

```

{
    ADD_SLOT(Array2D, my_array, INPUT, REQUIRED);
    ADD_SLOT(double, my_value, INPUT, 1.0);
public:
    inline void execute() override final
    {
        using onika::parallel::block_parallel_for;
        if( my_array->rows() == 0 || my_array->columns() == 0 ) { my_array->resize( 1024 , 1024 ); }
        BlockParallelValueAddFunctor value_add_func = { *my_array // reference our data array through its pointer and size
                                                         , *my_value // value to add to the elements of the array
                                                         };
        // Launching the parallel operation, which can execute on GPU if the execution context allows
        // result of parallel operation construct is captured into variable 'my_addition',
        // thus it can be scheduled in a stream queue for asynchronous execution rather than being executed right away
        auto my_addition = block_parallel_for( my_array->rows() // number of iterations, parallelize at the first level over rows
                                              , value_add_func // the function to call in parallel
                                              , parallel_execution_context("my_add_kernel") // execution environment inherited from this OperatorNode
                                              ); // optionally, we may tag here ^^^ parallel operation for debugging/profiling purposes
        // my_addition is scheduled here, transferring its content/ownership (see std::move) to the default stream queue
        auto stream_control = parallel_execution_stream() << std::move(my_addition) ;
        lout << "Parallel operation is executing..." << std::endl;
        stream_control.wait(); // wait for the operation to complete and results to be ready to read
        lout << "Parallel operation has completed !" << std::endl;
    }
};
}

```

If the execution context allows it, the parallel operation will proceed in the background, occupying either the free threads (other than those executing this code) or the *GPU*. This can be very useful, especially for overlapping computations and *MPI* message sends. The corresponding complete exemple in exaNBody source tree is here : `async_block_parallel.cpp`

Warning

The operation is asynchronous only if the execution context permits it. Otherwise, it will proceed synchronously and complete before the `block_parallel_for` function returns. In such cases, calling `control->wait()` will simply have no effect. When a parallel operation runs on the Cuda or HIP backend, asynchronous operations are always possible, thanks to the execution stream features supported by these backends. When a parallel operation runs on the OpenMP backend, real asynchronism depends on whether current executing `OperatorNode` is in a symmetric parallel or task based OpenMP context. The default behavior for an `OperatorNode` is to run in a symmetric parallel OpenMP context. This can be different when an encapsulating batch `OperatorNode` has been configured to switch to task mode OpenMP parallel execution (See batch configuration section).

14.1.3 Concurrent parallel executions

In the following example, we demonstrate how to run several parallel operations asynchronously, optionally running some of them concurrently.

```
namespace tutorial
{
    using namespace exanb;
    class ConcurrentBlockParallelSample : public OperatorNode
    {
        ADD_SLOT(Array2D, array1, INPUT_OUTPUT, REQUIRED);
        ADD_SLOT(Array2D, array2, INPUT_OUTPUT, REQUIRED);
        ADD_SLOT(double, value, INPUT, 1.0);
    public:
        inline void execute() override final
        {
            using onika::parallel::block_parallel_for;
            // if array1 is empty, allocate it
            if( array1->rows() == 0 || array1->columns() == 0 ) { array1->resize( 1024 , 1024 ); }
            // if array2 is empty, allocate it
            if( array2->rows() == 0 || array2->columns() == 0 ) { array2->resize( 1024 , 1024 ); }
            // functor to add the same value to all elements of array1
            BlockParallelValueAddFunctor array1_add_func = { *array1 // reference our data
            // array through its pointer and size
            , *value // value to add to the
            // elements of the array
            };
            // functor to add the same value to all elements of array2
            BlockParallelValueAddFunctor array2_add_func = { *array2, *value };
            // Launching the parallel operation, which can execute on GPU if the execution
            // context allows
            // result of parallel operation construct is captured into variable 'my_addition',
            // thus it can be scheduled in a stream queue for asynchronous execution rather
            // than being executed right away
            auto addition1 = block_parallel_for( array1->rows() // number of iterations,
            // parallelize at the first level over rows
            , array1_add_func // the function to call in
            // parallel
            , parallel_execution_context("add_kernel") //
```

(continues on next page)

(continued from previous page)

```

↪ execution environment inherited from this OperatorNode
                                ); // optionally, we may tag here ^^^ parallel_
↪ operation for debugging/profiling purposes
    // we create a second parallel operation we want to execute sequentially after the_
↪ first addition
    auto addition2 = block_parallel_for( array1->rows(), array1_add_func, parallel_
↪ execution_context("add_kernel") );
    // we finally create a third parallel operation we want to execute concurrently_
↪ with the two others
    auto addition3 = block_parallel_for( array2->rows(), array2_add_func, parallel_
↪ execution_context("add_kernel") );
    // addition1 and addition2 are scheduled asynchronously and sequentially one after_
↪ the other, int the stream queue #0
    auto stream_0_control = parallel_execution_stream(0) << std::move(addition1) <<_
↪ std::move(addition2) ;
    // addition3 is scheduled asynchronously in stream queue #1, thus it may run_
↪ concurrently with operations in stream quauae #0
    auto stream_1_control = parallel_execution_stream(1) << std::move(addition3) ;
    lout << "Parallel operations are executing..." << std::endl;
    stream_0_control.wait(); // wait for all operations in stream queue #0 to complete
    stream_1_control.wait(); // wait for all operations in stream queue #1 to complete
    lout << "All parallel operations have terminated !" << std::endl;
}
};
}

```

The corresponding complete exemple in exaNBody source tree is here : `concurrent_block_parallel.cpp`

14.2 Compute Cell Particles

This is the first specialized algorithmic pattern for particle systems. Therefore, it can only be applied to an instantiation of the *Grid* type from *exaNBody*. This pattern is the simplest; it will perform an independent operation in parallel on all particles in the system. To understand how this works in the implementation of a component, the following code presents a fully commented example aimed at increasing the velocity of all particles by a constant vector.

```

#include <exanb/core/operator.h>
#include <exanb/core/operator_slot.h>
#include <exanb/core/operator_factory.h>
#include <exanb/core/make_grid_variant_operator.h> // because we use make_grid_variant_
↪ operator for component registration
#include <exanb/compute/compute_cell_particles.h> // provides the compute_cell_
↪ particles function

namespace exaStamp {
    struct AddVec3Func // our functor, adds a vector to the 3 components of_
↪ particle velocity
    {
        const Vec3d vec_to_add; // functor parameter: the velocity to add

        ONIKA_HOST_DEVICE_FUNC
        inline void operator()(double& vx, double& vy, double& vz) const // parameters_
↪ defined by compute_fields_t declared below

```

(continues on next page)

(continued from previous page)

```

{
    vx += vec_to_add.x; // perform the desired operation
    vy += vec_to_add.y; // particle attributes are passed by reference (&), so they
    ↪ can be modified
    vz += vec_to_add.z; // here, we only need to implement the local operation on the
    ↪ particle, without worrying about thread block levels
}
};
}

namespace exanb {
    template<> struct ComputeCellParticlesTraits<exaStamp::AddVec3Functor>
    {
        static inline constexpr bool RequiresBlockSynchronousCall = false; // no
        ↪ collaboration between threads of the same block
        static inline constexpr bool CudaCompatible = true; // compatible
        ↪ with Cuda (thanks to ONIKA_HOST_DEVICE_FUNC usage)
    };
}

namespace exaStamp
{
    template< class GridT // our
    ↪ component adapts to any grid type
        , class = AssertGridHasFields<GridT,field::_vx,field::_vy,field::_vz> > // as
    ↪ long as particles have the vx, vy, and vz attributes
    class ParticleVelocityAdd : public OperatorNode
    {
        ADD_SLOT(Vec3d, vec, INPUT, REQUIRED); // vector to add to particle velocities
        ADD_SLOT(GridT, grid, INPUT_OUTPUT); // grid containing cells and all
        ↪ particles within the sub-domain
    public:
        inline void execute() override final {
            using compute_fields_t = FieldSet<field::_vx, field::_vy, field::_vz>; // fields on
            ↪ which our functor operates
            AddVec3Functor func = { *vec }; // instantiate
            ↪ the functor with the velocity provided as input to the component
            compute_cell_particles(*grid // grid containing particles
                                , false // do not apply our function in
            ↪ ghost zones
                                , func // the functor to apply
                                , compute_fields_t{} // attributes on which to compute
            ↪ defines the functor's call parameters
                                , parallel_execution_context() // component's parallel
            ↪ execution context
                                );
        }
    };
    // component registration
    template<class GridT> using ParticleVelocityAddTmpl = ParticleVelocityAdd<GridT>;
    CONSTRUCTOR_FUNCTION {
        OperatorNodeFactory::instance()->register_factory("add_velocity", make_grid_variant_

```

(continues on next page)

(continued from previous page)

```

↪operator<ParticleVelocityAddTpl>;
    }
}

```

Note

To improve the performance of Compute Cell Particles, you can choose to run it only on non-empty cells, meaning cells that contain at least one particle. This feature is particularly useful in DEM (Discrete Element Method), where it is common to encounter a significant number of empty cells. To use this feature, you can rely on the default parameters *filled_cells*, which is a list containing the indexes of the non-empty cells, and *number_filled_cells*, which indicates the number of non-empty cells.

14.3 Reduce Cell Particles

This algorithm pattern is very similar to the previous one, but it allows for computing the reduction of a calculation result across all particles. As we will see in the commented example below, the usage of this construction will be very similar to *compute_cell_particles* with one important exception: the functor can be called in multiple ways (multiple implementations of the operator () are possible). This reflects the need to perform reduction in three stages for performance reasons: local reduction within each thread, reduction of partial sums computed by threads within the same thread block, and finally the overall reduction that sums the partial contributions from different thread blocks.

```

#include <exanb/core/operator.h>
#include <exanb/core/operator_slot.h>
#include <exanb/core/operator_factory.h>
#include <exanb/core/make_grid_variant_operator.h> // because we use make_grid_variant_
↪operator for component registration
#include <exanb/compute/reduce_cell_particles.h> // provides the reduce_cell_particles_
↪function

namespace exaStamp {
    struct ReduceVec3NormFunctor // our functor calculates the sum of norms of forces
    {
        ONIKA_HOST_DEVICE_FUNC // operator for local_
        ↪reduction within a thread
        inline void operator ()( double & sum_norm // reference to accumulate_
        ↪contributions
            , double fx, double fy, double fz // particle force,_
        ↪parameters determined by reduce_fields_t declared below
            , reduce_thread_local_t // phantom parameter to_
        ↪differentiate call forms (here local thread reduction)
            ) const {
            sum_norm += sqrt( fx*fx + fy*fy + fz*fz ); // compute norm and add_
        ↪contribution
        }

        ONIKA_HOST_DEVICE_FUNC // operator for internal reduction_
        ↪within a thread block to a single block value
        inline void operator ()( double& sum_norm // reference to accumulate_
        ↪contributions from threads within the block
            , double other_sum // one of the partial sums to_

```

(continues on next page)

(continued from previous page)

```

→accumulate
        , reduce_thread_block_t // indicates block-level reduction
        ) const {
    ONIKA_CU_ATOMIC_ADD( sum_norm , other_sum ); // atomic addition function (thread-
→safe), works in both CUDA and CPU context
    }

    ONIKA_HOST_DEVICE_FUNC // operator for reduction across all
→thread blocks
    inline void operator ()( double& sum_norm // reference to global result
        , double other_sum // contribution from one block to add
        , reduce_global_t // indicates final reduction
        ) const {
    ONIKA_CU_ATOMIC_ADD( sum_norm , other_sum );
    }
};
}

namespace exanb {
    template< struct ReduceCellParticlesTraits<exaStamp::ReduceVec3NormFunctor>
    {
        static inline constexpr bool RequiresBlockSynchronousCall = false; // does not use
→intra-block thread collaboration
        static inline constexpr bool RequiresCellParticleIndex = false; // no additional
→parameters (cell/particle indices)
        static inline constexpr bool CudaCompatible = true; // CUDA compatible
    };
}

namespace exaStamp {
    template< class GridT // our
→component adapts to any grid type
        , class = AssertGridHasFields<GridT,field::_fx,field::_fy,field::_fz> > // as
→long as particles have fx, fy, and fz attributes
    class SumForceNorm : public OperatorNode
    {
        ADD_SLOT(GridT, grid, INPUT, REQUIRED); // grid containing cells and all
→particles within the sub-domain
        ADD_SLOT(double, sum_norm, OUTPUT); // output value = sum of force norms
    public:
        inline void execute() override final {
            using reduce_fields_t = FieldSet<field::_fx, field::_fy, field::_fz>; // fields on
→which our functor operates
            ReduceVec3NormFunctor func = {}; // instantiate
→the functor for summing force norms
            *sum_norm = 0.0;
            reduce_cell_particles(*grid // grid containing particles of
→the system
                , false // do not compute in ghost zones
                , func // functor to use
                , *sum_norm // initial value for reduction
→input, final result output

```

(continues on next page)

(continued from previous page)

```

        , reduce_fields_t{}           // fields used for reduction,
↪ defines call parameters
        , parallel_execution_context() // current component's parallel
↪ execution context
    );
}
};

// component registration similar to the one in the example for compute_cell_particles
}

```

This code defines a component (`SumForceNorm`) that computes the sum of norms of forces acting on particles within a grid (*grid*). Key points to note:

- It uses a functor (`ReduceVec3NormFunctor`) with multiple operator overloads to perform reduction across particles in three stages: local to each thread, within each thread block, and globally across all thread blocks.
- Traits (`ReduceCellParticlesTraits`) are specialized to specify that the functor supports *CUDA* and does not require intra-block thread synchronization.
- The component is registered using `OperatorNodeFactory`, making it accessible under a specific name for instantiation.

This example demonstrates advanced parallel computation techniques within a particle system framework (*exaNBody*), focusing on efficient reduction operations across large datasets.

Note

To improve the performance of Reduce Cell Particles, you can choose to run it only on non-empty cells, meaning cells that contain at least one particle. This feature is particularly useful in DEM (Discrete Element Method), where it is common to encounter a significant number of empty cells. To use this feature, you can rely on the default parameters *cells_idx*, which is a list containing the indexes of the non-empty cells, and *n_cells*, which indicates the number of non-empty cells.

14.4 Compute Pair Interaction

Pattern Overview:

- Purpose: Compute interactions (potentials or forces) between pairs of particles based on their proximity ($r_{ij} < r_{cut}$). - Components: Utilizes both particle grids and neighbor lists (see neighbor lists section). - Two Invocation Modes:
 - With Buffer: Collects attributes of neighboring particles into a buffer before invoking the user-defined functor once with this buffer.
 - Without Buffer: Computes interactions “on-the-fly” without pre-collecting neighbors’ attributes.
- Usage Examples: - Scenario 1: Interaction potentials require knowledge of all neighboring particles of at least one of the particles involved. Here, *compute_cell_particle_pairs* first accumulates neighbors’ attributes into a buffer and then invokes the user functor. - Scenario 2: Interaction potentials only require attributes of the two particles involved, making it efficient to compute interactions without using an intermediate buffer.
- Flexibility: - Functors can be implemented to support both invocation protocols (with and without buffer), allowing the subsystem to choose the optimal method based on execution environment and computational requirements. - Functors can choose to implement only one of these protocols if the scenario allows for a straightforward

implementation.

- Implementation: - The pattern involves defining a functor (`ComputePairInteractionFunctor`) that encapsulates the logic for computing interactions between pairs of particles. - Traits (`ComputeCellParticlePairsTraits`) are used to specify compatibility with *CUDA* and whether synchronous block calls are required.

Header files:

```
#pragma xstamp_enable_cuda           // Enable compilation with nvcc,
↳ allowing GPU code generation
#include <exanb/core/operator.h>      // Base to create a component
#include <exanb/core/operator_factory.h> // For registering the component
#include <exanb/core/operator_slot.h> // For declaring component slots
#include <exanb/core/make_grid_variant_operator.h> // For registering a template,
↳ component

#include <exanb/core/grid.h>          // Defines the Grid type containing
↳ particles
#include <exanb/core/domain.h>        // Domain type, represents the
↳ simulation domain
#include <exanb/particle_neighbors/chunk_neighbors.h> // GridChunkNeighbors type lists
↳ of neighbors
#include <exanb/compute/compute_cell_particle_pairs.h> // For compute_cell_particle_pairs,
↳ function
```

Functor Creation:

For this example (lennard jones potential), we implement both possible forms of invocation so that our functor can be used with or without an intermediate buffer. This allows `compute_cell_particle_pairs` to choose the most efficient method based on the context. In practice, you can implement only one of these forms (if your potential formulation allows) or both-either way, it will work on both *CPU* and *GPU*.

```
namespace microStamp {
    using namespace exanb;

    struct LennardJonesForceFunctor {
        const LennardJonesParms m_params; // parameters of our interaction potential
        // Operator for using an intermediate buffer containing all neighbors
        template<class ComputePairBufferT, class CellParticlesT>
        ONIKA_HOST_DEVICE_FUNC
        inline void operator () (
            size_t n, // number of neighboring particles
            const ComputePairBufferT& buffer, // intermediate buffer containing
↳ neighbor information
            double& e, // reference to energy, where we
↳ accumulate energy contribution
            double& fx, double& fy, double& fz, // references to force components
↳ where we add interaction contribution
            CellParticlesT* cells // array of all cells, in case
↳ additional information is needed
        ) const {
            double _e = 0.0; // local contributions, initialized to 0
            double _fx = 0.0;
            double _fy = 0.0;
```

(continues on next page)

(continued from previous page)

```

double _fz = 0.0;

for (size_t i = 0; i < n; ++i) {           // loop over neighbors in the buffer
    const double r = std::sqrt(buffer.d2[i]);
    double pair_e = 0.0, pair_de = 0.0;

    lj_compute_energy(m_params, r, pair_e, pair_de); // calculate energy and its
    ↪derivative

    const auto interaction_weight = buffer.nbh_data.get(i);
    pair_e *= interaction_weight;           // weight the interaction and
    ↪normalize over distance rij
    pair_de *= interaction_weight / r;

    _fx += pair_de * buffer.drx[i];         // add contributions from the i-
    ↪th neighbor
    _fy += pair_de * buffer.dry[i];
    _fz += pair_de * buffer.drz[i];
    _e += 0.5 * pair_e;
}

e += _e; // add local contributions to energy and force fields of the central
    ↪particle
fx += _fx;
fy += _fy;
fz += _fz;
}

// Operator for without buffer: one call per neighboring particle
template<class CellParticlesT>
ONIKA_HOST_DEVICE_FUNC
inline void operator () (
    ↪particle Vec3d dr,                       // relative position of the neighboring
    double d2,                             // square of the distance to the neighboring
    ↪particle double& e, double& fx, double& fy, double& fz, // references to variables to
    ↪update with energy and force contributions of the pair
    CellParticlesT* cells,                  // array of all cells, in case additional
    ↪information is needed
    size_t neighbor_cell,                  // index of the cell where the neighboring
    ↪particle resides
    size_t neighbor_particle,              // index of the neighboring particle within
    ↪its cell
    double interaction_weight              // weighting to apply on the interaction
) const {
    const double r = sqrt(d2);
    double pair_e = 0.0, pair_de = 0.0;

    lj_compute_energy(m_params, r, pair_e, pair_de); // calculate energy and its
    ↪derivative
    pair_e *= interaction_weight;           // weight and normalize by

```

(continues on next page)

(continued from previous page)

```

↪distance
    pair_de *= interaction_weight / r;

    fx += pair_de * dr.x;                                // add contributions of the pair_
↪to energy and forces of the central particle
    fy += pair_de * dr.y;
    fz += pair_de * dr.z;
    e += 0.5 * pair_e;
}
};
}

```

Define the compile-time characteristics of the functor:

```

namespace exanb
{
    template<> struct ComputePairTraits< microStamp::LennardJonesForceFunctor > //_
    ↪specialization for our functor
    {
        static inline constexpr bool RequiresBlockSynchronousCall = false; // no_
        ↪collaboration between threads within a block
        static inline constexpr bool ComputeBufferCompatible = true; // allows invocation_
        ↪with a compute buffer
        static inline constexpr bool BufferLessCompatible = true; // allows invocation_
        ↪without a buffer, for each neighbor
        static inline constexpr bool CudaCompatible = true; // compatible with Cuda
    };
}

```

Performing the calculation in parallel within the execute method of our component:

```

template<class GridT,                                // our_
↪component adapts to any type of grid
    class=AssertGridHasFields<GridT,field::_ep,field::_fx,field::_fy,field::_fz> > //_
↪which particles have attributes ep, fx, fy, and fz
class LennardJonesForce : public OperatorNode {
    ADD_SLOT( LennardJonesParms, config, INPUT, REQUIRED); // parameters_
    ↪of our potential
    ADD_SLOT( double, rcut, INPUT, 0.0); // cutoff_
    ↪radius
    ADD_SLOT( exanb::GridChunkNeighbors, chunk_neighbors, INPUT, exanb::GridChunkNeighbors
    ↪{}); // neighbor lists
    ADD_SLOT( bool, ghost, INPUT, false); // indicates_
    ↪if we calculate in ghost zones
    ADD_SLOT( Domain, domain, INPUT, REQUIRED); // simulation_
    ↪domain
    ADD_SLOT( GridT, grid, INPUT_OUTPUT ); // grid_
    ↪containing particles in the local subdomain
    using ComputeBuffer = ComputePairBuffer2<>; // shortcut_
    ↪for the type "buffer containing neighbors"
    using CellParticles = typename GridT::CellParticles; // shortcut_
    ↪for the type "grid cell"

```

(continues on next page)

(continued from previous page)

```

using NeighborIterator = exanb::GridChunkNeighborsLightWeightIt<false>; // shortcut
↪for the type "neighbor iterator"
using ComputeFields = FieldSet<field::_ep    // type defining the list of fields on
↪which our functor operates                ,field::_fx    // this list determines part of the
↪functor's call parameters                ,field::_fy    // we declare ep, fx, fy, and fz, which
↪are of type double, so we will find      ,field::_fz>; // references to these attributes as
↪parameters of our functor.
public:
    inline void execute () override final
    {
        auto optional = make_compute_pair_optional_args( // builds the list of
↪particle traversal options                NeighborIterator{ *chunk_neighbors } // iterator over neighbor
↪lists                                   , ComputePairNullWeightIterator{} // iterator over weighting
↪values (no weighting here)                , LinearXForm{ domain->xform() } // transformation to apply
↪on particle positions (domain deformation here) , ComputePairOptionalLocks<false>{}); // accessor to particle
↪access locks (no locking here)            compute_cell_particle_pairs( *grid // grid
↪containing particles                    , *rcut // cutoff
↪radius                                , *ghost // whether to
↪calculate in ghost zones or not          , optional // particle
↪traversal options                        , make_compute_pair_buffer<ComputeBuffer>() // creates
↪neighbor buffers                        , LennardJonesForceFunctor{ *config } //
↪instantiation of our functor with user parameters , ComputeFields{} // attributes
↪on which our functor operates            , DefaultPositionFields{} // uses
↪default attributes (rx, ry, and rz) for particle positions , parallel_execution_context() ); // parallel
↪execution context of the current component
    }
};

```


LIST OF PLUGINS

15.1 DefBox

15.1.1 Operator a

15.1.2 Operator b

15.1.3 Operator c

15.2 Core features

Independently from any loaded plugins, a set of core feature are always available for use. This includes, but is not limited to, domain description, particle grid, mpi core features, and more.

15.2.1 Domain

The domain struct holds information about simulation domain size and properties. It is initialized by an operator named “domain” in the YAML input file. Main domain properties are :

- domain lower and upper bounds
- cell size
- cell grid dimensions
- boundary condtions

Here after is an exemple of domain initialization using the “domain” operator :

```
domain:
  cell_size: 10.0 ang
  bounds: [ [ 0.0 , 0.0 , 0.0 ] , [ 10.0 , 10.0 , 10.0 ] ]
  periodic: [false,true,true]
  mirror: [ X- ]
```

Periodic indicates which directions have periodic boundary conditions. Mirror tells wich sides have mirror (reflective) boundary conditions. A reflective mirror side can be applied on any side of the domain box. A direction with at least one mirror boundary cannot be periodic, and vice-versa. As an exemple, if X direction is not periodic, it can have mirror condition and the lower end (X-) and/or the upper end (X+). If user set X to be periodic and add a X-/X+ mirror, X periodicity is disabled automatically. Finally, mirror property is a list of sides on which mirror conditions are applied, among X-, X+, X, Y-, Y+, Y, Z-, Z+ and Z. mirror sides X is a shortcut for enabling X- and X+ at the same time, so is Y and Z.

15.3 Logic

15.4 AMR

15.5 ParticleNeighbors

15.6 GridCellParticles

15.6.1 Cartesian Field Grid

- Operator Name: `set_cell_values`
- Description: This operator initializes values of a specific cell value field, and creates it if needed. optionally, initialization can be bounded to a specified region, the rest of field being set to all 0. This operator can also be used to refine the grid.
- Parameters:
 - `region`: Region of the field where the value is to be applied.
 - `grid_subdiv`: Number of (uniform) subdivisions required for this field. Note that the refinement is an octree.
 - `field_name`: Name of the field.
 - `value`: List of the values affected to the field.

YAML example:

```
- set_cell_values:
  field_name: jet
  grid_subdiv: 30
  value: [1, 0, 0, 20]
  region: GEYSERE
```

15.7 IO

15.8 Extra Storage

The purpose of this plugin is to add dynamic storage of any type associated with a particle. This package provides the data structure for storage, the associated MPI buffers and operators to move data between grid cells (move particle and migrate cell particles). This package was designed to meet the needs of exaDEM, as friction force values must be preserved from one time step to another. Additionally, this package allows for additional storage in the dump.

To utilize this additional storage, you need to use the data structure *ExtraDynamicDataStorageCellMoveBufferT* with your data type and the slot name *ges* for *grid extra storage*. This structure is a grid of *CellExtraDynamicDataStorageT*, which itself contains an array of your data type (`m_data`) and a particle information array (`m_info`) containing the start index in the `m_data` array, the number of stored data, and the particle index.

```
template<typename ItemType> struct CellExtraDynamicDataStorageT
using UIntType = uint64_t;
using InfoType = ExtraStorageInfo;
onika::memory::CudaMMVector<InfoType> m_info; /**< Info vector storing indices of the
↪ [start, number of items, particle id] of each cell's extra dynamic data in m_data. */
onika::memory::CudaMMVector<ItemType> m_data; /**< Data vector storing the extra dynamic
↪ data for each cell. */
```

With the data structure *ExtraStorageInfo*:

```
using UIntType = uint64_t;
UIntType offset; ///< start / offset
UIntType size;   ///< number of items
uint64_t pid;    ///< Particle Id
```

The filling of these data structures is your responsibility; however, it is possible to instantiate an operator to verify that the filling has been done correctly:

```
namespace exanb
{
    template<class GridT> using CheckInfoConsistencyYourDataTypeTpl = CheckInfoConsistency
    <GridT, GridExtraDynamicDataStorageT<YOUR_DATA_TYPE>>;

    // === register factories ===
    CONSTRUCTOR_FUNCTION
    {
        OperatorNodeFactory::instance()->register_factory( "check_es_consistency_your_data_
        type", make_grid_variant_operator< CheckInfoConsistencyYourDataTypeTpl > );
    }
}
```

The following code is an example of how to correctly fill the data structure (type = Interaction):

```
typedef GridExtraDynamicDataStorageT<Interaction> GridCellParticleInteraction;
ADD_SLOT( GridT, grid, INPUT_OUTPUT , REQUIRED );
ADD_SLOT( GridCellParticleInteraction , ges , INPUT_OUTPUT );
```

```
auto& g = *grid;
const auto cells = g.cells();
const size_t n_cells = g.number_of_cells(); // nbh.size();
auto & ces = ges->m_data;
assert( ces.size() == n_cells );
const IJK dims = g.dimension();
const int gl = g.ghost_layers();

#pragma omp parallel
{
    Interaction item;
    GRID_OMP_FOR_BEGIN(dims-2*gl,_,block_loc, schedule(guided) )
    {
        IJK loc_a = block_loc + gl;
        size_t cell_a = grid_ijk_to_index( dims , loc_a );
        const unsigned int n_particles = cells[cell_a].size();
        auto& storage = ces[cell_a];
        auto& data = storage.m_data;
        auto& info = storage.m_info;
        // auto& history = extract_history(data);
        // You can extract data before initialize.
        storage.initialize(n_particles);
        for(size_t i = 0 ; i < n_particles ; i++)
        {
            // Do some stuff and fill item.
        }
    }
}
```

(continues on next page)

(continued from previous page)

```

    // You can add several items here.
    auto& [offset, size, id] = info[i];
    size++;
    m_data.push_back(item);
    // you can update the particle offset here.
}
}
GRID_OMP_FOR_END
// you can fit offsets here instead of in the omp loop. (offset(i) = offset(i-1) +
↪size(i-1))
}

```

Warning:

- This package allows for as many external storages as there are types; however, it's not possible to have two additional storages of the same type.
- Don't forget to adjust the size of this storage to the number of cells in the grid when first using it.
- This package does not integrate with routines for particle-level calculations such as *compute_cell_particles*.

Tip:

- Before sending or writing data, consider removing unnecessary information. For example, in DEM, if the friction is equal to (0,0,0), you can overwrite this data to save space. (more details, see in exaDEM *compress_interaction* operator).

15.8.1 Extra Data Checker

- Operator: *check_es_consistency_double*
 - *Description* : This operator checks if for each particle information the offset and size are correct
 - *ges* : Your grid of additionnal data storage.

YAML example:

```
check_es_consistency_double
```

15.8.2 Migrate Cell Particles With Extra Storage

- Operator: *migrate_cell_particles_double* (example)
 - *Description* : *migrate_cell_particles* does 2 things:
 - * 1. it repartitions the data accross mpi processes, as described by *lb_block*.
 - * 2. it reserves space for ghost particles, but do not populate ghost cells with particles. The ghost layer thickness (in number of cells) depends on *ghost_dist*. Inputs from different mpi process may have overlapping cells (but no duplicate particles). the result grids (of every mpi processes) never have overlapping cells. The ghost cells are always empty after this operator.
 - *ges* : Your grid of additionnal data storage.
 - *bes* : Your buffer used for particles moving outside the box
 - *buffer_size* : Performance tuning parameter. Size of send/receive buffers in number of particles.
 - *copy_task_threshold* : Performance tuning parameter. Number of particles in a cell above which an asynchronous OpenMP task is created to pack particles to send buffer.

- *extra_receive_buffers*: Performance tuning parameter. Number of extraneous receive buffers allocated allowing for asynchronous (OpenMP task) particle unpacking. A negative value *n* is interpreted as $-n \times \text{NbMpiProcs}$
- *force_lb_change*: Force particle packing/unpacking to and from send buffers even if a load balancing has not been triggered
- *otb_particles*: Particles outside of local processor's grid
- In practice, do not tune this operator yourself.

How to create your operator:

```
#include <exanb/extra_storage/migrate_cell_particles_es.hpp>
namespace exanb
{
    template<class GridT> using MigrateCellParticlesYourDataTypeTpl =
    ↪ MigrateCellParticlesES<GridT, GridExtraDynamicDataStorageT<your_data_type>>;

    // === register factory ===
    CONSTRUCTOR_FUNCTION
    {
        OperatorNodeFactory::instance()->register_factory( "migrate_cell_particles_your_data_
    ↪ type", make_grid_variant_operator<MigrateCellParticlesYourDataTypeTpl> );
    }
}
```

YAML example:

```
migrate_cell_particles_double
```

15.8.3 Move Particles With Extra Storage

- Operator: *migrate_cell_particles_double* (example)
 - *Description*: This operator moves particles and extra data storage (es) across cells.
 - *ges*: Your grid of additionnal data storage.
 - *bes*: Your buffer used for particles moving outside the box
 - *otb_particles*: Particles outside of local processor's grid
 - In practice, do not tune this operator yourself

How to create your operator:

```
#include <exanb/extra_storage/move_particles_es.hpp>
namespace exanb
{
    template<class GridT> using MoveParticlesYourDataTypeTpl = MigrateCellParticlesWithES
    ↪ <GridT, GridExtraDynamicDataStorageT<your_data_type>>;

    // === register factory ===
    CONSTRUCTOR_FUNCTION
    {
        OperatorNodeFactory::instance()->register_factory( "migrate_cell_particles_your_data_
    ↪ type", make_grid_variant_operator<MoveParticlesYourDataTypeTpl> );
    }
```

(continues on next page)

(continued from previous page)

```

}
}

```

YAML example:

```
move_particles_double
```

15.8.4 IO Writer With Extra Data

There is no operator in exaNBody for writing dump files with storage because you need to explicitly specify the fields to store. However, we propose a non-instantiated templated operator for this purpose. We provide an example with exaDEM and Interaction data type.

```

#include <exaDEM/interaction/grid_cell_interaction.hpp>
#include <exanb/extra_storage/sim_dump_writer_es.hpp>
#include <exanb/extra_storage/dump_filter_dynamic_data_storage.h>

namespace exaDEM
{
    using namespace exanb;
    using DumpFieldSet = FieldSet<field::_rx,field::_ry,field::_rz, field::_vx,field::_vy,
    ↪ field::_vz, field::_mass, field::_homothety, field::_radius, field::_orient , field::_
    ↪ mom , field::_vrot , field::_arot, field::_inertia , field::_id , field::_shape >;

    template<typename GridT> using SimDumpWriteParticleInteractionTpl =
    ↪ SimDumpWriteParticleES<GridT, exaDEM::Interaction, DumpFieldSet>;

    // === register factories ===
    CONSTRUCTOR_FUNCTION
    {
        OperatorNodeFactory::instance()->register_factory( "write_dump_particle_interaction"
    ↪ , make_grid_variant_operator<SimDumpWriteParticleInteractionTpl> );
    }
}

```

For the description of operator slots, see `write_dump_particle_interaction` in exaDEM documentation. Tip: compress extra storage before write dump data file.

YAML example:

```

dump_data_particles:
- timestep_file: "exaDEM_%09d.dump"
- message: { mesg: "Write dump " , endl: false }
- print_dump_file:
    rebind: { mesg: filename }
    body:
        - message: { endl: true }
- compress_interaction
- stats_interactions
- write_dump_particle_interaction
- chunk_neighbors_impl

```

15.8.5 IO Reader With Extra Data

There is no operator in exaNBody for reading dump files with storage because you need to explicitly specify the fields to store. However, we propose a non-instantiated templated operator for this purpose. We provide an example with exaDEM and Interaction data type.

```
#include <exaDEM/interaction/grid_cell_interaction.hpp>
#include <exanb/extra_storage/sim_dump_reader_es.hpp>

namespace exaDEM
{
    using namespace exanb;
    using DumpFieldSet = FieldSet<field::_rx,field::_ry,field::_rz, field::_vx,field::_vy,
    ↪ field::_vz, field::_mass, field::_homothety, field::_radius, field::_orient , field::_
    ↪ mom , field::_vrot , field::_arot, field::_inertia , field::_id , field::_shape >;

    template<typename GridT> using SimDumpReadParticleInteractionTpl =
    ↪ SimDumpReadParticleES<GridT, exaDEM::Interaction, DumpFieldSet>;

    // === register factories ===
    CONSTRUCTOR_FUNCTION
    {
        OperatorNodeFactory::instance()->register_factory( "read_dump_particle_interaction" ,
    ↪ make_grid_variant_operator<SimDumpReadParticleInteractionTpl> );
    }
}
```

For the description of operator slots, see *read_dump_particle_interaction* in exaDEM documentation.

YAML example:

```
read_dump_particle_interaction:
  filename: last.dump
  override_domain_bounds: false
  #scale_cell_size: 0.5
```

15.9 MPI

15.9.1 Update Ghost Layers

- Operator: *ghost_update_r* and *ghost_update_all*

- *Description* : These operators are in charge of updating ghost zones between two sub-domains and copying the information required at sub-domains boundaries and for periodic conditions. The *ghost_update_r* operator copies the position while *ghost_update_all* copies all fields defined in your grid type.
- *gpu_buffer_pack* : boolean value [false] to decide if you want to port pack/unpack routines on GPU.
- *async_buffer_pack* : boolean value [false] triggering to overlap several calls to pack and unpack (send buffers as soon as possibles).
- *staging_buffer* : boolean value [false] triggering the copy to a pure CPU buffer before MPI calls (highly recommended if packaging on GPU)
- *serialize_pack_send* : boolean value [false] triggering to wait that all send buffers are built up before sending the first one.

Example in your msp file:

```
- ghost_update_r:  
  gpu_buffer_pack: true  
  async_buffer_pack: true  
  staging_buffer: true
```

Note that you can customize a *ghost_update_XXX* operator for your application such as :

```
namespace exaDEM  
{  
    using namespace exanb;  
    using namespace UpdateGhostsUtils;  
    // === register factory ===  
    template<typename GridT> using UpdateGhostsYourFields = UpdateGhostsNode< GridT ,  
↪ FieldSet<field::_rx, field::_ry, field::_rz , list_of_your_fields > , false >;  
  
    CONSTRUCTOR_FUNCTION  
    {  
        OperatorNodeFactory::instance()->register_factory( "ghost_update_XXX", ↪  
↪ make_grid_variant_operator<UpdateGhostsYourFields> );  
    }  
}
```

15.9.2 MPI Barrier

This operator is used to create synchronization points between MPI processes. In practice, it is utilized to obtain accurate timing information from operators during performance studies. Otherwise, timing accumulate in operators containing MPI collective routines such as *displ_over*.

- Operator : *mpi_barrier*
 - *Description* : Add a MPI_Barrier(MPI_COMM_WORLD).
 - *mpi* : MPI_Comm, default is MPI_COMM_WORLD

YAML Example:

```
- mpi_barrier
```


EXANBODY COMMAND LINES

ExaNBody offers a range of command-line options to facilitate debugging, profiling, and configuring OpenMP settings, providing users with enhanced control and flexibility over the behavior and performance of the code. These options enable developers to diagnose issues, analyze performance characteristics, and optimize parallel execution on multi-core architectures. From enabling debug mode to specifying thread affinity and scheduling policies, users can tailor ExaNBody's behavior to suit their specific requirements and achieve optimal performance. This section outlines the various command-line options available, empowering users to leverage ExaNBody's capabilities effectively.

16.1 Command line and input file interaction

exaNBody based apps treat command line the same way as input files, the command line just being YAML elements expressed with another syntax. YAML document, as processed by an exaNBody app is formed by a set of included files (either implicitly or explicitly through the **includes** list) and the user input file passed as the first argument of command line, as in the following example :

```
./exaStamp myinput.msp
```

The YAML document built from user input and its included files have 3 reserved dictionary entries, namely **configuration** reserved for configuring execution sub-system, **includes** reserved to include other YAML files and **simulation** which is interpreted as the the root batch operator representing the whole simulation to run.

When it comes to interpreting command line arguments, the exaNBody based application parses it and internally converts it to a YAML document, merged with previously read YAML inputs (user input file and included files). Conversion from command line argument to YAML proceeds as follows :

- If argument starts with **-set-**, it is understood as a generic YAML dictionary entry, and each '-' is interpreted as a marker for an inner dictionary entry. The following value is understood as being the value associated with the given dictionary key. If no value found after a **-set-xxx** style argument, the value **true** is implicitly used. For instance, **-set-global-rcut_inc '1.2 m'** is equivalent to including a YAML file which contains

```
global:  
  rcut_inc: 1.2 m
```

- If a command line argument starts with **-xxx** with xxx being anything but 'set' , then similar rules apply as for the **-set-xxx** args, but the YAML block is understood as being a dictionary entry inside the **configuration** block. For instance, **-logging-debug true** is equivalent to inclusion of a YAML file containing :

```
configuration:  
  logging:  
    debug: true
```

16.2 Commands ‘Help’ for your application

exaNBody provides several help commands to assist users in understanding the available command lines, plugins, and operators. Below are the descriptions and usage of each help command.

16.2.1 General Help Command

To display the general help information:

```
./exaDEM --help
```

This command shows the available help command lines.

16.2.2 Command-Line Options Help

To display the command line options available:

```
./exaDEM --help command-line
```

This command lists the command line options you can use to customize the configuration block. See the next section named “Index of command line options to customize configuration block” for detailed information.

16.2.3 Plugins Help

To print all operators and plugins available in your application:

```
./exaDEM --help plugins
```

This command prints every operator and plugin available in *exaNBody*.

16.2.4 Show Plugins Help

To show detailed information for every operator, including descriptions and slot operators:

```
./exaDEM --help show-plugins
```

This command provides detailed information for all operators, including their descriptions and slot operators.

16.2.5 Operator-Specific Help

To show details for a specific operator, including its description and slot operators:

```
./exaDEM --help [operator-name]
```

Replace *[operator-name]* with the name of the operator you want details for.

16.3 Index of command line options to customize configuration block

Option	Value type	Default
–logging-parallel	bool	false
–logging-debug	bool	false
–logging-log_file	std::string	“”

continues on next page

Table 1 – continued from previous page

Option	Value type	Default
–logging-err_file	std::string	“”
–logging-dbg_file	std::string	“”
–profiling-resmem	bool	false
–profiling-exectime	bool	false
–profiling-summary	bool	false
–profiling-filter	StringVector	{ }
–profilingtrace-enable	bool	false
–profilingtrace-format	std::string	“yaml”
–profilingtrace-file	std::string	“trace”
–profilingtrace-color	std::string	“operator”
–profilingtrace-total	bool	false
–profilingtrace-idle	bool	true
–profilingtrace-trigger	std::string	“”
–profilingtrace-trigger_interval	IntVector	{ }
–profilingtrace-idle_resolution	long	8192
–profilingtrace-idle_smoothing	long	32
–debug-plugins	bool	false
–debug-config	bool	false
–debug-yaml	bool	false
–debug-graph	bool	false
–debug-ompt	bool	false
–debug-graph_addr	bool	false
–debug-graph_lod	int	1
–debug-graph_fmt	std::string	“console”
–debug-graph_rsc	bool	false
–debug-files	bool	false
–debug-rng	std::string	“”
–debug-graph_filter	StringVector	{ }
–debug-filter	StringVector	{ }
–debug-particle	UInt64Vector	{ }
–debug-particle_nbh	bool	false
–debug-particle_ghost	bool	false
–debug-fpe	bool	false
–debug-verbose	int	0
–debug-graph_exec	bool	false
–onika-parallel_task_core_mult	int	ONIKA_TASKS_PER_CORE
–onika-parallel_task_core_add	int	0
–onika-gpu_sm_mult	int	ONIKA_CU_MIN_BLOCKS_PER_SM
–onika-gpu_sm_add	int	0
–onika-gpu_block_size	int	ONIKA_CU_MAX_THREADS_PER_BLOCK
–onika-gpu_disable_filter	StringVector	{ }
–nogpu	bool	false
–mpimt	bool	true
–pinethreads	bool	false
–threadrotate	int	0
–omp_num_threads	int	-1
–omp_max_nesting	int	-1
–omp_nested	bool	false
–plugin_dir	std::string	USTAMP_PLUGIN_DIR
–plugin_db	std::string	“”

continues on next page

Table 1 – continued from previous page

Option	Value type	Default
<code>--plugins</code>	StringVector	<code>{ }</code>
<code>--generate_plugins_db</code>	bool	false
<code>--help</code>	std::string	""
<code>--run_unit_tests</code>	bool	false
<code>--set</code>	YAML::Node	

16.4 Tune your run with OpenMP

Harnessing the power of OpenMP parallelization, ExaNBody provides users with the ability to fine-tune their execution environment for optimal performance on multicore architectures. Through a selection of command-line options, users can customize thread management, affinity settings, and loop scheduling to maximize parallel efficiency. This subsection introduces the command-line options available for configuring OpenMP behavior within ExaNBody.

Table 2: ExaNBody OpenMP Command Lines

Type of tools	Command line	Description	Default
Pine OMP Threads	<code>--pinthreads true</code>	Controls thread affinity settings within the OpenMP runtime, influencing how threads are bound to CPU cores for improved performance, particularly on NUMA architectures.	false
Set the num- ber of threads	<code>--omp_num_threads 10</code>	Specifies the number of threads to be utilized for parallel execution, allowing users to control the degree of parallelism based on system resources and workload characteristics.	By default it takes the maximum number of threads available
Maximum level of nested parallelism	<code>--omp_max_nesting [max_nesting_level]</code>	Specifies the maximum level of nested parallelism allowed within OpenMP, controlling the depth at which parallel regions can be nested.	-1
Nested paral- lelism within OpenMP	<code>--omp_nested [true/false]</code>	Enables or disables nested parallelism within OpenMP, allowing parallel regions to spawn additional parallel regions. true may be replaced by an integer indicating how many nested parallelism levels are allowed.	false
MPI_THREAD_ feature request	<code>--mpimt [true/false]</code>	If set to true, requires that MPI implementation supports MPI_THREAD_MULTIPLE feature	true

16.5 Tune GPU execution options

Harnessing the power of GPU parallelization, ExaNBody provides users with the ability to fine-tune their execution environment for optimal performance on GPU accelerators. Through a selection of command-line options, users can customize GPU configuration management.

Table 3: ExaNBody GPU Command Lines

Type of tools	Command line	Description	Default
disable GPU	<code>--nogpu</code>	disables use of GPU accelerators, even though some are available.	false
workgroup / block size	<code>--onika-gpu_block_size N</code>	sets default thread block size to N.	128
selectively disable GPU	<code>--onika-gpu_disable_filter ["sim.loop.compute_fc", ".*hook_force"]</code>	a list of regular expressions matching operator paths for which GPU execution will be disabled	[]

16.6 Profiling tools available in exaNBody

ExaNBody offers a comprehensive suite of performance profiling tools designed to empower users in analyzing and optimizing their parallel applications. These tools provide valuable insights into runtime behavior, resource utilization, and performance bottlenecks, enabling developers to fine-tune their code for maximum efficiency. From CPU profiling to memory analysis, ExaNBody's profiling tools offer a range of capabilities to meet diverse profiling needs. This section introduces the profiling tools available within ExaNBody, equipping users with the means to gain deeper understanding and enhance the performance of their parallel applications. some profiling tools are accessible through configuration flags, others need to insert specific operator nodes in simulation description.

Table 4: ExaNBody Profiling Tools Command Lines

Type of tools	Command line / Operator	Defaults	Description
Timers (configuration flag)	<code>--profiling-summary [true false]</code>	true	Displays timer information for every bottom level or batch operator nodes in simulation graph. if two percentages are shown, second one is the portion of time spent in the nearest enclosing loop batch operator (i.e. <code>compute_loop</code>)
VITE Trace (configuration flag)	<code>--profilingtrace-file [true false]</code>	""	Generates a VITE trace describing CPU threads occupation (not available GPU threads).
Resident memory (configuration flag)	<code>--profiling-resmem [true false]</code>	false	Displays resident memory evolution during the execution. Helps tracking of memory leaks.
Performance adviser (operator)	not present by default, can be inserted after initialization phase or after first iteration, i.e. using <code>+first_iteration: [performance_advisor]</code>	<code>performance_advisor: { verbose: true }</code>	Displays some tips about parameters to be tweaked to improve your simulations performance (cell size, number of MPI processes, etc.)

16.7 Using Timers with MPI and GPU

In ExaNBody, timers are essential tools for measuring performance in MPI and GPU-accelerated computations. This section explores their use within ExaNBody's parallel implementations, providing insights into runtime behavior and performance characteristics.

This tools provides the list of timers for every operators in a hierarchical form.

- Number of calls
- CPU Time
- GPU Time
- Imbalance time between mpi processes (average and maximum)
- execution time ratio

The Imbalance value is computed as : $I = (T_{max} - T_{ave}) / T_{ave} - 1$

With the variables:

- T_{max} is the execution time of the slowest MPI process.
- T_{ave} is the average time spent over MPI processes.
- I is the imbalance value.

Note that if you force to stop your simulation, the timer are automatically printed in your terminal.

Output with OpenMP:

Profiling	tot. time (GPU)	avginb	maxinb
↪ count percent			
sim	2.967e+04	0.000	0.000
↪ 1 100.00%			
... // some logs			
loop	2.964e+04	0.000	0.000
↪ 1 99.88%			
scheme	2.881e+04	0.000	0.000
↪ 100000 97.09%			
combined_compute_prolog	2.300e+03	0.000	0.000
↪ 100000 7.75%			
check_and_update_particles	1.016e+04	0.000	0.000
↪ 100000 34.25%			
particle_displ_over	2.154e+03	0.000	0.000
↪ 100000 7.26%			
update_particles_full	6.482e+03	0.000	0.000
↪ 5961 21.84%			
update_particles_full_body	6.474e+03	0.000	0.000
↪ 5961 21.82%			
compact_neighbor_friction	1.621e+02	0.000	0.000
↪ 5961 0.55%			
move_particles_friction	6.347e+02	0.000	0.000
↪ 5961 2.14%			
trigger_load_balance	2.591e+02	0.000	0.000
↪ 5961 0.87%			
trigger_lb_tmp	6.095e+00	0.000	0.000
↪ 5961 0.02%			
nth_timestep	3.342e+00	0.000	0.000

(continues on next page)

(continued from previous page)

→	5961	0.01%	extend_domain	2.389e+02	0.000	0.000	└
→	5961	0.80%					
...							

Output with MPI:

Profiling			tot. time (GPU)	avginb	maxinb	└
→	count	percent				
sim			2.376e+04	0.000	0.000	└
→	1	100.00%				
... // some logs						
loop			2.372e+04	0.000	0.000	└
→	1	99.82%				
scheme			2.308e+04	0.086	2.249	└
→	100000	97.13%				
combined_compute_prolog			5.779e+02	0.280	2.937	└
→	100000	2.43%				
check_and_update_particles			1.687e+04	0.454	2.770	└
→	100000	70.97%				
particle_displ_over			4.067e+03	0.687	2.643	└
→	100000	17.11%				
update_particles_full			1.159e+04	0.167	0.812	└
→	6001	48.78%				
update_particles_full_body			1.159e+04	0.167	0.813	└
→	6001	48.76%				
compact_neighbor_friction			7.170e+01	0.387	0.876	└
→	6001	0.30%				
move_particles_friction			1.797e+02	0.254	0.853	└
→	6001	0.76%				
trigger_load_balance			9.340e+01	0.674	1.787	└
→	6001	0.39%				
trigger_lb_tmp			2.582e+00	0.187	2.836	└
→	6001	0.01%				
nth_timestep						
extend_domain			8.655e+01	0.733	2.016	└
→	6001	0.36%				
...						

16.8 Debug features in exaNBody

ExaNBody is equipped with a range of debug features tailored to aid developers in the debugging process. This section outlines the comprehensive list of debug functionalities available within ExaNBody, providing developers with essential tools to diagnose and resolve issues effectively. This is an exhaustive list:

Table 5: ExaNBody Debug Command Lines

Type of tools	Command line	Description
Simulation graph	<code>-debug-graph [true false]</code>	Prints simulation graph as built after user input file, included files and command lines are processed
user input YAML document	<code>-debug-yaml [true false]</code>	Prints final YAML document used by application after user input file, included files and command lines are merged
Output ldbg	<code>-logging-debug true</code>	Print debug logs added in <i>ldbg</i> <<
filtering debug output	<code>-debug-filter ["regexp1","regexp2",...]</code>	Filters which operator nodes output debug messages with <i>ldbg</i> <<"...". <i>regexp</i> is a regular expression matching operator pathname, i.e. it's name within block and sub block, for instance "sim.first_iteration.compute_force.lj_force" can be filtered differently than <i>sim.compute_loop.compute_force.lj_force</i> . alternatively, adding a filter expression such as <i>".*lj_force"</i> will activate debug messages for all instances of <i>lj_force</i> operator.

How to use output *ldbg*:

Possibility to active it only for one operator:

- Command line : `-logging-debug true -debug-filter[".*operator1",".*operator2",...]`
- Operator name : logging and debug

Example in your input file (.msp):

```
configuration:
  logging: { debug: false , parallel: true }
  debug:
    filter: [ ".*init_neighbor_friction" , ".*move_particles_friction" , ".*check_nbh_
↪friction" , ".*compact_neighbor_friction" , ".*extend_domain" ]
```


17.1 Add a new plugin

You can create new plugins if you wish to develop a new set of operators that can be compiled independently of others or require specific plugin compilation. Here's how to define and add a new plugin.

First of all, add your new plugin directory in *src/CMakeLists.txt*:

```
mkdir my_new_plugin
add_subdirectory(my_new_plugin)
cd my_new_plugin
vi CMakeLists.txt
```

Create a *CMakeLists.txt* and use the macros defined to plug your operators to exaNBody support such as:

```
set(list_of_plugins_required exanbIO exanbDefBox exanbParticleNeighbors exadem_numerical_
↪scheme exadem_friction exadem_force_field)
set(exadem_my_new_plugin_LINK_LIBRARIES ${list_of_plugins_required})
xstamp_add_plugin(exadem_my_new_plugin ${CMAKE_CURRENT_SOURCE_DIR})
```

Warning: Do not forget to do a *make UpdatePluginDataBase*.

17.2 Add a new operator

Initially, it's crucial to establish a precise definition of the intended kernel, the targeted data, and the method for executing this kernel.

17.2.1 Define your kernel

We recommend writing the computation kernel (in a functor) in a *.h* file within an *include/exaDEM* folder of your plugin directory. For instance, if one intends to apply gravitational force to a particle, it's necessary to have knowledge of the external force and access to mutators for modifying the force fields. The kernel for a particle is then written in a file *include/exaDEM/gravity_force.h*:

```
namespace exaDEM
{
    struct GravityForceFunctor
    {
        exanb::Vec3d g = { 0.0 , 0.0 , -9.807};
        ONIKA_HOST_DEVICE_FUNC inline void operator () (double mass, double& fx, double& fy, ↪
        ↪double& fz ) const
```

(continues on next page)

(continued from previous page)

```

{
    fx += g.x * mass;
    fy += g.y * mass;
    fz += g.z * mass;
}
};
}

```

It is possible to provide specific kernel *Traits* that will be used by the data traversal functions (ex: *ComputeCellParticles*). For example, if we want to provide the GPU capabilities for this kernel:

```

namespace exanb
{
template<> struct ComputeCellParticlesTraits<exaDEM::GravityForceFunctor>
{
    static inline constexpr bool CudaCompatible = true;
};
}

```

17.2.2 Define an operator:

To begin with, all operators inherit from the *OperatorNode* class, where you establish the necessary physical fields for applying your operator. Following that, it's essential to define the slots that will establish the execution graph among operators. Ultimately, you are required to furnish an *Execute()* function, which the graph will execute, along with a *Documentation()* function.

This is an example of an empty operator:

```

#include <exaDEM/your_kernel.h>
namespace exaDEM
{
    using namespace exanb;
    template<typename GridT
        , class = AssertGridHasFields< GridT, list_of_fields>
        >
    class YourOperator : public OperatorNode
    {
        using ComputeFields = FieldSet< list_of_fields>;
        static constexpr ComputeFields compute_field_set {};
        ADD_SLOT( GridT , grid , INPUT_OUTPUT );
    public:

        inline std::string documentation() const override final
        {
            return R"EOF(Your operator.)EOF";
        }

        inline void execute () override final
        {
            YourFunctor func;
            compute_cell_particles( *grid , false , func , compute_field_set , gpu_execution_
↪ context() , gpu_time_account_func() );
        }
    }
}

```

(continues on next page)

(continued from previous page)

```
};

template<class GridT> using YourOperatorTpl = YourOperator<GridT>;

// === register factories ===
CONSTRUCTOR_FUNCTION
{
    OperatorNodeFactory::instance()->register_factory( "your_operator_name", make_grid_
    ↪variant_operator< YourOperatorTpl > );
}
}
```

17.2.2.1 Comments:

- Please refers to *src/forcefields/gravity_force.cpp* to illustrate a simple example.
- `AssertGridHasFields` allows to eliminate wrong grids
- If *ComputeFields* does not correspond to the input parameters of your functor, the operator can't compile.
- **ADD_SLOT macro works as *ADD_SLOT(Type, Name, TypeOfSlot, DefaultValue, Documentation)* with:**
 - Type: *double, int ...*
 - TypeOfSlot: *INPUT, OUTPUT* or *INPUT_OUTPUT*
 - Documentation: is a *DocString* type: *DocString{slot documentation}*
- To access to a slot value, please add `*`.
- Constant value are hidden the operator, for example the gravity functor is initialized as: *GravityForceFunctor func { *gravity};*, gravity is a slot.
- Note that you need to specify you operator for a given Grid and define the *operator_name* in your dictionary.

17.2.2.2 Comment:

You can do your own traversal function by explicitly iterating over cells. Example:

```
ADD_SLOT( GridT , grid , INPUT_OUTPUT );
inline void execute () override final
{
    MPI_Comm comm = *mpi;
    auto cells = grid->cells();
    IJK dims = grid->dimension();
    size_t ghost_layers = grid->ghost_layers();
    IJK dims_no_ghost = dims - (2*ghost_layers);
    # pragma omp parallel
    {
        GRID_OMP_FOR_BEGIN(dims_no_ghost,_,loc_no_ghosts)
        {
            IJK loc = loc_no_ghosts + ghost_layers;
            size_t cell_i = grid_ijk_to_index(dims,loc);
            auto& cell_ptr = cells[cell_i];
            const size_t n = cells[cell_i].size();
            auto* __restrict__ my_field = cell_ptr[field::my_field];
        }
    }
    # pragma omp simd
```

(continues on next page)

(continued from previous page)

```
    for(size_t j=0;j<n;j++)  
        _my_field[j] += 27;  
    }  
}  
}
```

EXANBODY PUBLICATIONS

Main citation:

Carrard, T., Prat, R., Latu, G., Babilotte, K., Lafourcade, P., Amarsid, L., & Soulard, L. (2023). ExaNBody: a HPC framework for N-Body applications. [1]

Field High Performance Computing:

Babilotte, K., Dubois, A., Carrard, T., Lafourcade, P., Videau, L., Molinari, J. F., & Soulard, L. (2025). On-the-fly clustering for exascale molecular dynamics simulations. *Computer Physics Communications*, 307, 109427. cite:*babilotte2025on*

Carrard, T., Prat, R., Latu, G., Babilotte, K., Lafourcade, P., Amarsid, L., & Soulard, L. (2023). ExaNBody: a HPC framework for N-Body applications. [1]

Prat, R., Carrard, T., Soulard, L., Durand, O., Namyst, R., & Colombet, L. (2020). AMR-based molecular dynamics for non-uniform, highly dynamic particle simulations. *Computer Physics Communications*, 253, 107177. [8]

Prat, R., Colombet, L., & Namyst, R. (2018, August). Combining task-based parallelism and adaptive mesh refinement techniques in molecular dynamics simulations. In *Proceedings of the 47th International Conference on Parallel Processing* (pp. 1-10). [9]

Dirand, E., Colombet, L., & Raffin, B. (2018). Tins: A task-based dynamic helper core strategy for in situ analytics. In *Supercomputing Frontiers: 4th Asian Conference, SCFA 2018, Singapore, March 26-29, 2018, Proceedings 4* (pp. 159-178). Springer International Publishing. [3]

Cieren, E., Colombet, L., Pitoiset, S., & Namyst, R. (2014). ExaStamp: A parallel framework for molecular dynamics on heterogeneous clusters. In *Euro-Par 2014: Parallel Processing Workshops: Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II 20* (pp. 121-132). Springer International Publishing. [2]

Field Physics / Molecular Dynamics:

Lafourcade, P., Ewald, G., Carrard, T., & Denoual, C. (2025). Extraction of slip systems and twinning variants from a Lagrangian analysis of molecular dynamics simulations. *Mechanics of Materials*, 200, 105189. []

Soulard, L., Durand, O., Burie, J. R., & Babilotte, K. (2024). Micro-jetting: Areal density calculation from a triangular groove. *Journal of Applied Physics*, 135(20). [13]

Soulard, L. (2023). Micro-jetting: A semi-analytical model to calculate the velocity and density of the jet from a triangular groove. *Journal of Applied Physics*, 133(8). [10]

Soulard, L., Carrard, T., & Durand, O. (2022). Molecular dynamics study of the impact of a solid drop on a solid target. *Journal of Applied Physics*, 131(13). [11]

Soulard, L., Durand, O., Prat, R., & Carrard, T. (2021). High velocity impact of a spherical particle on a surface: Theory and simulation of the jet formation. *Journal of Applied Physics*, 129(20).

Soulard, L., & Durand, O. (2020). Observation of phase transitions in shocked tin by molecular dynamics. *Journal of Applied Physics*, 127(16). [12]

Durand, O., Soulard, L., Colombet, L., & Prat, R. (2020). Influence of the phase transitions of shock-loaded tin on microjetting and ejecta production using molecular dynamics simulations. *Journal of Applied Physics*, 127(17). [4]

Durand, O., Jaouen, S., Soulard, L., Heuze, O., & Colombet, L. (2017). Comparative simulations of microjetting using atomistic and continuous approaches in the presence of viscosity and surface tension. *Journal of Applied Physics*, 122(13). [5]

EXAMPLE FOR RST USAGE (MATH, CODE ETC)

19.1 Example of math citatiion :

To cite paper, place the appropriate reference in the file source/Bibliography.rst. Then, you can cite it by using this kind of command:

My paper is written by (AaBbCcDd et al.)

19.2 Example of math with latex as equation :

$$\frac{\sum_{t=0}^N f(t, k)}{N} \quad (19.1)$$

19.3 Example of math with latex as inline text :

Quand $(a > 0)$ alors il y a deux solutions à $(ax^2 + bx + c = 0)$ et elles sont exactement $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$.

Le double dollar indique une équation “standalone” sur sa ligne

19.4 Example of bash code :

```
mkdir my_new_plugin
add_subdirectory(my_new_plugin)
cd my_new_plugin
vi CMakeLists.txt
```

19.5 Example of cpp code :

```
namespace exaSPH
{
    struct TestFunctor
    {
        exanb::Vec3d a = { 1.0 , 2.0 , 3.0};
        ONIKA_HOST_DEVICE_FUNC inline void operator () (double mass, double& fx, double& fy,
↪double& fz ) const
        {
            fx += a.x * mass;
```

(continues on next page)

(continued from previous page)

```
    fy += a.y * mass;  
    fz += a.z * mass;  
  }  
};  
}
```

19.6 Example of yaml code :

```
domain:  
  cell_size: 2 m  
  periodic: [false,true,false]
```

19.7 Example of simple figure :

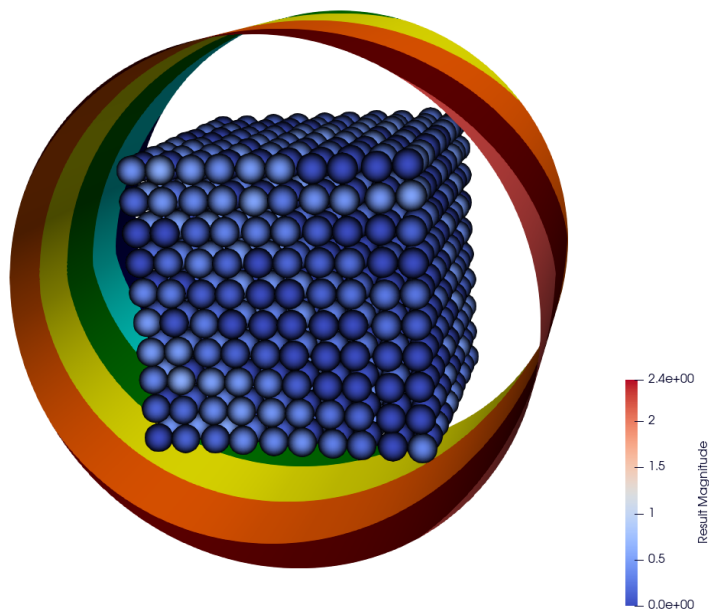


Fig. 1: Figure 1: This is the caption of the figure (a simple paragraph).

INSTALLING EXANB AND YOUR APPLICATION ON ADASTRA

Disclaimer: This section has been written the 02/07/24 and the environment may have changed in the meantime. This section gives some commands to help users to compile and use their application on Adastra. Example are done with *exaDEM*.²

20.1 Download sources

```
cd ${HOME}
mkdir sources; cd sources
(git clone https://github.com/Collab4exaNBody/exaDEM.git;
cd exaDEM;
git checkout xdm-parallel-execution-streaming)
(git clone https://github.com/Collab4exaNBody/exaNBody.git;
cd exaNBody;
git checkout parallel-execution-streaming)
SRC_DIR=${HOME}/sources/exaDEM
XNB_DIR=${HOME}/sources/exaNBody
```

20.2 Load your Environment:

```
PROJECT_SETUP_ENV_COMMANDS="module purge ; module load PrgEnv-gnu/8.5.0 ; module load
↳craype-x86-trento craype-accel-amd-gfx90a ; module load rocm/5.5.1 ; ml .omnitrace/1.
↳10.4 ; ml .omniperf/1.0.10"
eval ${PROJECT_SETUP_ENV_COMMANDS}
BUILD_DIR=${WORKDIR}/build/exaDEM
RUN_DIR=${WORKDIR}/bench
```

20.3 Configuration File Example

Put the following lines in your configure-exaDEM.sh file

```
cmake \
  -DCMAKE_C_COMPILER=hipcc \
  -DCMAKE_CXX_COMPILER=hipcc \
  -DCMAKE_CXX_FLAGS="-Wpass-failed" \
  -DCMAKE_HIP_FLAGS="-Wpass-failed" \
  -DXNB_BUILD_CUDA=ON \
  -DXNB_ENABLE_HIP=ON \
```

(continues on next page)

(continued from previous page)

```

-DCMAKE_HIP_PLATFORM=amd \
-DONIKA_HAVE_OPENMP_DETACH=OFF \
-DONIKA_HAVE_OPENMP_TOOLS=OFF \
-DONIKA_HIP_COMPILE_FLAGS="-Werror=return-stack-address;-Werror=return-type" \
-DOpenMP_CXX_LIB_NAMES="gomp;pthread" \
-DOpenMP_gomp_LIBRARY=/opt/rocm-5.5.1/llvm/lib/libgomp.so \
-DHOST_HW_CORES=192 \
-DHOST_HW_THREADS=384 \
-DCMAKE_HIP_ARCHITECTURES=gfx90a \
-DCMAKE_BUILD_TYPE=Release \
-DEXASTAMP_TEST_DATA_DIR=${HOME}/data \
-DPROJECT_SETUP_ENV_COMMANDS="${PROJECT_SETUP_ENV_COMMANDS}" \
-DCMAKE_INSTALL_PREFIX=${HOME}/install/exaDEM \
-Dyaml-cpp_DIR=/lus/home/CT6/cad14959/tcarrard/install/yaml-cpp/lib64/cmake/yaml-
↪cpp \
-DexaNBody_DIR=${XNB_DIR} \
${SRC_DIR}

```

Comment: You will need to specify your YAML-CPP installation.

20.4 Compilation:

```

source configure-exaDEM.sh
mkdir -p $BUILD_DIR
rm -rf $BUILD_DIR/*
cd $BUILD_DIR

```

20.5 Compile And Run Your Application On Adastr

Get a node:

```

salloc --reservation=HackathonGPU -n 1 --account=cad14959 --gpus-per-node=8 --nodes=1 --
↪cpus-per-task=32 --constraint=MI250 --time 2:00:0

```

Compile your code:

```

srun --cpus-per-task 32 make -j 32
make UpdatePluginDataBase

```

Run your code:

```

mkdir ${RUN_DIR}; cd ${RUN_DIR}
cp <ROOT>/input.msp .
srun ${BUILD_DIR}/exaDEM input.msp

```

20.6 Profiling tools:

Rocprof:

```
srun rocprof --stats --hsa-trace --hip-trace --basenames off --timestamp on -o genesis_${SLURM_JOBID}.${SLURM_PROCID}.csv "$@" ../exaDEM input.msp
```

Omniperf: Run and Analyze

```
srun omniperf profile -n name_run --roof-only -- binary ../exaDEM input.msp  
omniperf analyze -p workloads/name_run/mi200 > omniperf_analyze.out
```

Omnitrace:

```
srun omnitrace-avail -G omnitrace.cfg --all  
export OMNITRACE_CONFIG_FILE=./omnitrace.cfg  
srun omnitrace-sample ../exaDEM input.msp
```


BIBLIOGRAPHY

- [1] Thierry Carrard, Raphaël Prat, Guillaume Latu, Killian Babilotte, Paul Lafourcade, Lhassan Amarsid, and Laurent Soulard. ExaNBody: a HPC framework for N-Body applications. In *Euro-Par 2023: Parallel Processing Workshops*, 342–354. Cham, 2024. Springer Nature Switzerland. doi:10.1007/978-3-031-50684-0_27.
- [2] Emmanuel Cieren, Laurent Colombet, Samuel Pitoiset, and Raymond Namyst. Exastamp: a parallel framework for molecular dynamics on heterogeneous clusters. In *Euro-Par 2014: Parallel Processing Workshops: Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II 20*, 121–132. Springer, 2014. doi:10.1007/978-3-319-14313-2_11.
- [3] Estelle Dirand, Laurent Colombet, and Bruno Raffin. Tins: a task-based dynamic helper core strategy for in situ analytics. In *Supercomputing Frontiers: 4th Asian Conference, SCFA 2018, Singapore, March 26-29, 2018, Proceedings 4*, 159–178. Springer, 2018. doi:10.1007/978-3-319-69953-0_10.
- [4] O Durand, L Soulard, L Colombet, and R Prat. Influence of the phase transitions of shock-loaded tin on microjetting and ejecta production using molecular dynamics simulations. *Journal of Applied Physics*, 2020. doi:10.1063/5.0003744.
- [5] Olivier Durand, S Jaouen, L Soulard, Olivier Heuze, and Laurent Colombet. Comparative simulations of microjetting using atomistic and continuous approaches in the presence of viscosity and surface tension. *Journal of Applied Physics*, 2017. doi:10.1063/1.4994789.
- [6] Luning Fang, Ruochun Zhang, Colin Vanden Heuvel, Radu Serban, and Dan Negrut. Chrono:: gpu: an open-source simulation package for granular dynamics using the discrete element method. *Processes*, 9(10):1813, 2021.
- [7] Marc Josien and Raphaël Prat. Parallel and bias-free rsa algorithm for maximal poisson-sphere sampling. *Computer Physics Communications*, 305:109354, 2024. doi:https://doi.org/10.1016/j.cpc.2024.109354.
- [8] Raphaël Prat, Thierry Carrard, Laurent Soulard, Olivier Durand, Raymond Namyst, and Laurent Colombet. Amr-based molecular dynamics for non-uniform, highly dynamic particle simulations. *Computer Physics Communications*, 253:107177, 2020. doi:10.1016/j.cpc.2020.107177.
- [9] Raphaël Prat, Laurent Colombet, and Raymond Namyst. Combining task-based parallelism and adaptive mesh refinement techniques in molecular dynamics simulations. In *Proceedings of the 47th International Conference on Parallel Processing*, 1–10. 2018. doi:10.1145/3225058.3225085.
- [10] L Soulard. Micro-jetting: a semi-analytical model to calculate the velocity and density of the jet from a triangular groove. *Journal of Applied Physics*, 2023. doi:10.1063/5.0142057.
- [11] L Soulard, Th Carrard, and O Durand. Molecular dynamics study of the impact of a solid drop on a solid target. *Journal of Applied Physics*, 2022. doi:10.1063/5.0083266.
- [12] Laurent Soulard and O Durand. Observation of phase transitions in shocked tin by molecular dynamics. *Journal of Applied Physics*, 2020.
- [13] Laurent Soulard, Olivier Durand, Jean-René Burie, and Killian Babilotte. Micro-jetting: areal density calculation from a triangular groove. *Journal of Applied Physics*, 2024. doi:10.1063/5.0209692.