Documentation

This package contains several python codes to run simulations of rigid bodies made out of *blob particles* rigidly connected near a single wall (floor). These codes can compute the mobility of complex shape objects, solve mobility or resistance problems for suspensions of many bodies or run deterministic or stochastic dynamic simulations.

We explain in the next sections how to use the package. For the theory consult the references:

1. **Brownian Dynamics of Confined Rigid Bodies**, S. Delong, F. Balboa Usabiaga and A. Donev. The Journal of Chemical Physics, **143**, 144107 (2015). DOI arXiv
2. **Hydrodynamics of suspensions of passive and active rigid particles: a rigid multiblob approach**, F. Balboa Usabiaga, B. Kallemov, B. Delmotte, A. Pal Singh Bhalla, B. E. Griffith and A. Donev. arXiv
3. **Brownian Dynamics of Active Sphere Suspensions Confined Near a No-Slip Boundary**, F. Balboa Usabiaga, B. Delmotte and A. Donev, submitted to The Journal of Chemical Physics.

Note: The key blob-blob translational mobility in the presence of a wall is computed here using the Rotne-Prager-Blake tensor in the appendices B and C of: **Simulation of hydrodynamically interacting particles near a no-slip boundary**, James Swan and John Brady, Phys. Fluids **19**, 113306 (2007)DOI. We modify the mobility to allow overlaps between blobs and between blobs and the wall, see Ref. [3].

1. Prepare the package

The codes are implemented in python (version 2.x) and it is not necessary to compile the package to use it. However, we provide alternative implementations in *C++* (through the Boost Python library) and *pycuda* for some of the most computationally expensive functions. You can skip to section 2 but come back if you want to take fully advantage of this package.

1.1 Prepare the mobility functions

The codes use functions to compute the blob mobility matrix **M** and the matrix vector product **Mf**. For some functions we provide a *C++* implementation which can be around five times faster than the python version. We also provide *pycuda* implementations which, for large systems, can be orders of magnitude faster. To use the *C++* implementation move to the directory `mobility/` and compile

`mobility_ext.cc` to a `.so` file using the Makefile provided (which you will need to modify slightly to reflect your Python version, etc.).

To use the *pycuda* implementation all you need is *pycuda* and a GPU compatible with CUDA; you don't need to compile any additional file in this package.

1.2 Blob-blob forces

In dynamical simulations it is possible to include blob-blob interactions to, for example, simulate a colloid suspension with a given steric repulsion. Again, we provide versions in *python*, *C++* and *pycuda*. To use the *C++* version move to the directory `multi_bodies/` and compile `forces_ext.cc` to a `.so` file using the Makefile provided (which you will need to modify slightly to reflect your Python version, etc.).

To use the *pycuda* implementation all you need is *pycuda* and a GPU compatible with CUDA; you don't need to compile any additional file in this package.

2. Rigid bodies configuration

We use a vector (3 numbers) and a quaternion (4 numbers) to represent the location and orientation of each body, see Ref. 1 for details. This information is saved by the code in the `*.clones` files, with format:

```
number_of_rigid_bodies
vector_location_body_0 quaternion_body_0
vector_location_body_1 quaternion_body_1
.
.
.
```

For example, the file `multi_bodies/Structures/boomerang_N_15.clones` represents the configuration of a single body with location (0, 0, 10) and orientation given by the quaternion (0.5, 0.5, 0.5, 0.5).

The coordinates of the blobs forming a rigid body in the default configuration (location (0, 0, 0) and quaternion (1, 0, 0, 0)) are given to the codes through `*.vertex` files. The format of these files is:

```
number_of_blobs_in_rigid_body
vector_location_blob_0
vector_location_blob_1
.
.
.
```

For example, the file `multi_bodies/Structures/boomerang_N_15.vertex` gives the structure of a boomerang-like particle formed by 15 blobs.

3. Active slip

The blobs can have an active slip as described in the Ref. 2, therefore, we can simulate the dynamics of active bodies like bacteria or self-propelled colloids. The code assigns a slip function to each body depending on its structure ID; the structure ID is the name of the `*.clones` file without the *path* or the ending *.clones* (i.e., *boomerang_N_15* for the file `multi_bodies/Structures/boomerang_N_15.clones`). Therefore all the bodies passed to the code in the same `.clones` file will have the same slip. This way it is easy to combine active and passive bodies in the same simulation by just using different `.clones` files for active and passive bodies.

Right now, the code only has two active slip functions implemented; for bodies with structure ID *active_body* all blobs have a slip along the x-axis in the reference configuration. For bodies with any other structure ID the slip is set to zero, i.e., they are passive bodies. It is easy to generalize the code to include other kind of slips, see Section 5.3 for details.

4. Run static simulations

We start explaining how to compute the mobility of a rigid body close to a wall. First, move to the directory `multi_bodies/` and inspect the input file `inputfile_body_mobility.dat`:

```
# Select problem to solve
scheme                                    body_mobility

# Select implementation to compute the blobs mobility
mobility_blobs_implementation             python

# Set fluid viscosity (eta) and blob radius
eta                                       1.0
blob_radius                               0.25

# Set output name
output_name                               data/run.body_mobility

# Load rigid bodies configuration, provide
# *.vertex and *.clones files
structure   Structures/boomerang_N_15.vertex Structures/boomerang_N_15.clones
```

Now, to run the code, use

```
python multi_bodies_utilities.py --input-file inputfile_body_mobility.dat
```

The code writes the results in files with the prefix `data/run.body_mobility`. You can inspect the file `*.body_mobility.dat` to see the 6x6 rigid body mobility.

List of input file options:

- `scheme`: (string) Options: `mobility, resistance and body_mobility`. Select the problem to solve. `mobility` computes the velocities of a suspension of rigid bodies subject to external forces and torques (see below). `resistance` computes the forces and torques on rigid bodies moving with given velocities (see below). `body_mobility` computes the mobility of one rigid body as in the above example.
- `mobility_blobs_implementation`: Options: `python and C++`. It selects which implementation is used to compute the blob mobility matrix $\mathbf{M}$. See section 1 to use the C++ version.
- `mobility_vector_prod_implementation`: Options: `python, C++ and pycuda`. It selects the implementation to compute the matrix vector product $\mathbf{Mf}$. See section 1 to use the C++ or pycuda implementations.
- `eta`: (float) the fluid viscosity.
- `blob_radius`: (float) the hydrodynamic radius of the blobs.
- `solver_tolerance`: (float) the tolerance for the iterative mobility solver.
- `output_name`: (string) the prefix used to save the output files.
- `velocity_file`: (string) name of a file with the velocities of the rigid bodies used in the `resistance` problem. The format of the file is one line per body and six floats per line corresponding to linear (first three) and angular velocities (last three).
- `force_file`: (string) name of a file with the forces and torques used in the `mobility` problem. The format of the file is one line per body and six floats per line corresponding to force (first three) and torque (last three). If no file is given the code compute the forces on the bodies as explained in the section 5.2.
- `structure`: (two strings) name of the vertex and clones files with the rigid bodies configuration, see section 2. To simulate bodies with different shapes add to the input file one `structure` option per each kind of body and give their `vertex` and `clones` files, see multibodies/inputfile.dat for an example.

- `plot_velocity_field`: (x_0 x_1 N_x y_0 y_1 N_y z_0 z_1 N_z) if the code is run with this options and the schemes `mobility` or `resistance` the code plots the velocity field of the fluid to a `vtk` file. The velocity field is plotted in a rectangular box with the lower corner located at `(x_0, y_0, z_0)`, the upper corner located at `(x_1, y_1, z_1)` and using a grid of dimensions `(N_x, N_y, N_z)`. The `vtk` file can be postprocessed with external software like `VisIt` from the Lawrence Livermore National Laboratory or `ParaView` from Sandia National Laboratories to generate an image of the velocity field. See figure 1 as an example.

- `tracer_radius`: (float (default 0)) effective radius of the nodes where the fluid velocity field is computed with the option `plot_velocity_field`. The default value, zero, computes the pointwise fluid velocity, a larger value average the fluid velocity over a region of radius `tracer_radius`.

The output files are:

- `.inputfile`: It is a copy of the input file to better keep track of input options.
- `.body_mobility.dat`: the 6x6 rigid body mobility computed with the option `scheme body_mobility`. When this mobility is apply to the vector (force, torque) generates the vector (velocity, angular_velocity).
- `.force.dat`: forces and torques on the bodies computed with the option `scheme resistance`. The format of the file is one line per body and six floats per line corresponding to force (first three) and torque (last three).
- `.velocities.dat`: velocities of the rigid bodies computed with the option `scheme mobility`. The format of the file is one line per body and six floats per line corresponding to linear (first three) and angular velocities (last three).

5. Run dynamic simulations

5.1 Rigid multiblob simulations

We have two python codes to run dynamic simulations. The first, in the directory `boomerang/`, allows to run stochastic Brownian simulations for a single body. See the instruction in `doc/boomerang.txt`. Here, we explain how to use the other code which allows to run deterministic and stochastic simulations for many bodies.

First, move to the directory `multi_bodies/` and inspect the input file `inputfile_dynamic.dat`:

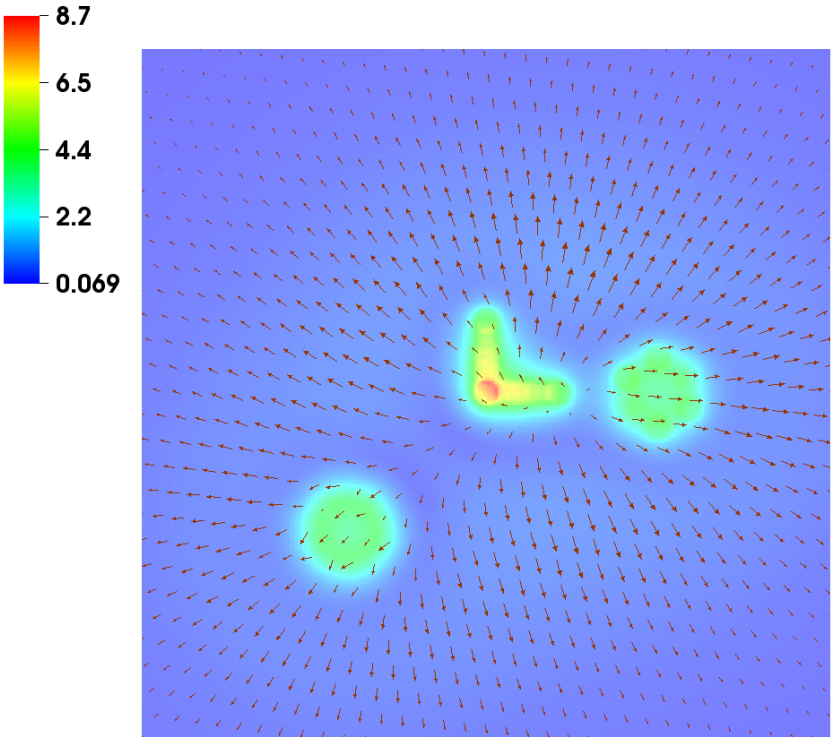

Figure 1: Velocity field around two spheres and a boomerang shaped colloid falling towards a bottom wall. Velocity field computed with the option `plot_velocity_field` and image generated with the software VisIt.

```
# Select integrator
scheme                                  deterministic_adams_bashforth

# Select implementation to compute M and M*f
mobility_blobs_implementation           python
mobility_vector_prod_implementation     python

# Select implementation to compute the blob-blob interactions
blob_blob_force_implementation          python

# Set time step, number of steps and save frequency
dt                                      0.1
n_steps                                 10
n_save                                  1

# Set fluid viscosity (eta), gravity (g) and blob radius
eta                                     1.0
g                                       1.0
blob_radius                             0.25

# Set parameters for the blob-blob interation
repulsion_strength                      1.0
debye_length                            1.0

# Set interaction with the wall
repulsion_strength_wall                 1.0
debye_length_wall                       1.0

# Set output name
output_name                             data/run

# Load rigid bodies configuration, provide
# *.vertex and *.clones files
structure Structures/boomerang_N_15.vertex Structures/boomerang_N_15.clones
structure Structures/shell_N_12_Rg_1.vertex Structures/shell_N_12_Rg_1.clones
```

With this input we can run a simulation with three rigid bodies, one with a boomerang shape and two with a spherical shape; see structures given to the options `structure`. To run the simulation use

```
python multi_bodies.py --input-file inputfile_dynamic.dat
```

Now, you can inspect the output, `ls data/run.*`. The output files are:

- `.bodies_info`: it contains the number of bodies and blobs in the simulation. Also how many types of bodies (i.e. different shapes) and how many bodies of each type.
- `.clones`: For each time step saved and each structure type the code saves a file with the location and orientation of the rigid bodies. The name format is (output_name + structure_name + time_step + .clones) The format of the files is the same that in the input .clones files.
- `.inputfile`: a copy of the input file.
- `.random_state`: it saves the state of the random generator at the start of the simulation. It can be used to run a simulation with the same random numbers.
- `.time`: the wall-clock time elapsed during the simulation (in seconds).

List of options for the input file:

- `scheme`: (string) Options: `deterministic_forward_euler_dense_algebra, deterministic_forward_euler, deterministic_adams_bashforth, stochastic_first_order_RFD, stochastic_first_order_RFD_dense_algebra and stochastic_adams_bashforth`. It selects the scheme to solve the mobility problem and integrate the equation of motion. The `*forward_euler*` schemes are first order accurate while `*adams_bashforth*` are second order accurate in the deterministic case. The scheme `*dense_algebra` uses dense algebra methods to solve the mobility problem and therefore the computational cost scales like (number_of_blobs)**3. The other schemes use preconditioned GMRES to solve the mobility problem and are more efficient for large number of bodies.
- `mobility_blobs_implementation`: Options: `python and C++`. This option indicates which implementation is used to compute the blob mobility matrix **M**. See section 1 to use the C++ version.
- `mobility_vector_prod_implementation`: Options: `python, C++, pycuda and pycuda_single`. This option select the implementation to compute the matrix vector product **Mf**. See section 1 to use the C++ or pycuda implementation. The option `pycuda_single` uses single precision (it is faster in GPUs) the others use double precision.
- `blob_blob_force_implementation`: Options: `None, python, C++ and pycuda`. Select the implementation to compute the blob-blob interactions between all pairs of blobs. If None is selected the code does not compute blob-blob interactions. The cost of this function scales like (number_of_blobs)**2, just like the product **Mf**.

- `body_body_force_torque_implementation`: Options: `None and python`. Select the implementation to compute the body-body interactions between all pairs of bodies. This function provides and alternative way to compute force between bodies without iterating over all the blobs in the system. Note that this force will be added to the forces coming from other terms like blob-blob interactions. If None is selected the code does not compute body-body interactions directly but it can compute blob-blob interactions which lead to effective body-body interactions. The cost of this function scales like (number_of_bodies)**2. See Section 5.3 for more details.

- `eta`: (float) the fluid viscosity.

- `blob_radius`: (float) the hydrodynamic radius of the blobs.

- `solver_tolerance`: (float) the relative tolerance for the iterative mobility solver.

- `output_name`: (string) the prefix used to save the output files.

- `dt`: (float) time step length to advance the simulation.

- `n_steps`: (int) number of time steps.

- `initial_step`: (int (default 0)) Use this option to restart a simulation. If `initial_step > 0` the code will run from time step `initial_step` to `n_steps`. Also, the code will try to load `.clones` files with the name (output_name + structure_name + initial_step + .clones).

- `n_save`: (int) save the bodies configuration every `n_save` steps.

- `repulsion_strength`: (float) the blobs interact through a soft potential of the form (`U = eps + eps * (d-r)/b` if `r < d` and `U = eps * exp(-(r-d)/b)` if `r >=d`) where `r` is the distance between blobs, `b` is the characteristic length, `eps` is the strength and `d=2*a` is twice the blob radius. This is the strength of the potential, `eps` in the above expression (see section 5.3 to modify blobs interactions).

- `debye_length`: (float) the blobs interact through a potential (`U = eps + eps * (d-r)/b` if `r < d` and `U = eps * exp(-(r-d)/b)` if `r >=d`), this is the characteristic length of the potential, `b` in the above expression (see section 5.3 to modify blobs interactions).

- `repulsion_strength_wall`: (float) the blobs interact with the wall with a soft potential. The potential is (`U = eps + eps * (d-r)/b` if `r < d` and `U = eps * exp(-(r-d)/b)` if `r >=d`) where `h` is the distance between the wall and the particle, `d=a` is the blob radius, `b` is the characteristic potential length and `eps` is the strength. This is the strength of the Yukawa potential, `eps` in the above formula (see section 5.3 to modify blobs interactions).

- `debye_length_wall`: (float) the blobs interact with the wall with a Yukawa-like potential (`U = eps + eps * (d-r)/b` if `r < d` and `U = eps * exp(-(r-d)/b)` if `r >=d`). This is the characteristic length of the Yukawa potential, `b` in the above expression (see section 5.3 to modify blobs interactions).
- `random_state`: (string) name of a file with the state of the random generator from a previous simulation. It can be used to generate the same random numbers in different simulations.
- `seed`: (unsigned int) seed for the random number generator. It is not used if a `random_state` is defined in the input file. If neither `seed` nor `random_state` are defined the code will automatically initialized to a pseudorandom state (see documentation for numpy.random.RandomState).
- `structure`: (two strings) name of the vertex and clones files with the rigid bodies configuration, see section 2. To simulate bodies with different shapes add to the input file one `structure` option per each kind of body.
- `save_clones`: (string (default one_file_per_step)) options *one_file_per_step* and *one_file*. With the option *one_file_per_step* the clones configuration are saved in one file per kind of structure and per time step as explained above. With the option *one_file* the code saves one file per kind of structure with the configurations of all the time steps; configurations of different time steps are separated by a line with the number of rigid bodies.
- `periodic_length`: (three floats (default 0 0 0)) length of the unit cell along the x, y and z directions. If the length of the unit cell along the x or y directions is larger than zero the code uses Pseudo Periodic Boundary Conditions (PPBC) along that axis, otherwise the system is considered infinite in that direction. With PPBC, particles forces are computed using the minimum image convention as with standard periodic boundary conditions. Hydrodynamic interactions are computed between particles in the unit cell and the first neighbor cells along the pseudo-periodic axis. PPBC along the z axis are not supported.

5.2 Rollers simulations

We can also use the code `multi_bodies.py` to run simulations of bodies discretized with a single blob interacting hydrodynamically with a grand-mobility matrix that includes couplings between the linear and angular velocities, see Ref. [3] for a detailed description. To run a simulation use:

```
python multi_bodies_utilities.py --input-file inputfile_body_mobility.dat
```

The input file options are the same than for a rigid multiblob simulation (see section 5.1 and the file `inputfile_body_mobility.dat`) except for the following differences:

- `scheme`: (string) Options: `deterministic_forward_euler_rollers`, `stochastic_first_order_rollers`, `deterministic_adams_bashforth_rollers`, `stochastic_adams_bashforth_rollers`, `stochastic_mid_point_rollers`, `stochastic_trapezoidal_rollers`. We provide several schemes for deterministic and stochastic simulations.

- `structure`: (two strings) name of the vertex and clones files with the rigid bodies configuration, see section 2. However, this code only accepts bodies discretized with a single blob so the vertex file is trivial, see file `multi_bodies/Structures/blob.vertex`.

- `free_kinematics`: (string (default True)) if `free_kinematics` is True the angular velocity of the blobs is not fixed and each blob is subject to a torque `T=8*pi*eta*a^3*omega_one_roller` (see below). If `free_kinematics` is False all the blobs rotate with a prescribed angular velocity given with the option `omega_one_roller` but they are free to translate. The torque acting on the blobs is a Lagrangian multiplier that enforces the prescribed angular velocity.

- `omega_one_roller` (three floats (default 0 0 0)) prescribed angular velocity of the blobs if the option `free_kinematics` is set to False. If `free_kinematics` is set to True the blobs are subject to a constant torque `T=8*pi*eta*a^3*omega_one_roller`.

5.3 Modify the codes

Right now, the slip on the rigid bodies and the interactions between blobs and between
bodies are hard-coded in the codes. We explain here how the user can change these functions. First, we provide two alternatives to compute the interactions between bodies. A direct method that uses the locations and orientations of the bodies and a indirect form that compute the forces between all the blobs forming the rigid bodies and then it uses those forces to compute the forces and torques on the bodies as explaned in the Refs 1 and 2. The second approach can be more expensive, since in general `number_of_blobs >> number_of_bodies` but it can be used with bodies with arbitrary shapes, while the first method it is hard to generalize to non-spherical bodies. Note that the code can used both methods at the same time. You can modify the following functions:

- blob-blob interactions: to modify the *python* implementation edit the function `blob_blob_force` in the file `multi_bodies/multi_bodies_functions.py`. To modify the *C++* implementation edit the function `blobBlobForce` in the file `multi_bodies/forces_ext.cc` and recompile the *C++* code. To modify the *pycuda* version edit the function `blob_blob_force` in the file `multi_bodies/forces_pycuda.py`

- blob-wall interaction: to modify the *python* implementation edit the function `blob_external_force` in the file `multi_bodies/multi_bodies_functions.py`. There are not *C++* or *pycuda* versions of this function since it is not an expensive operation.
- body-body interactions: to modify the *python* implementation edit the function `body_body_force_torque` in the file `multi_bodies/multi_bodies_functions.py`.
- body external forces: to modify the one-body forces, for example gravity or interactions with the wall, edit the function `bodies_external_force_torque` in the file `multi_bodies/multi_bodies_functions.py`.
- active slip: The code assigns a slip function to each body depending on its structure ID; the structure ID is the name of the `*.clones` file without the *path* or the ending *.clones* (i.e., *boomerang_N_15* for the file `multi_bodies/Structures/boomerang_N_15.clones`). Therefore all the bodies passed to the code in the same `.clones` file will have the same slip. The user can generalize the slip functions by editing the function `set_slip_by_ID` in the file `multi_bodies/multi_bodies_functions.py`. We provide an example of how to add a constant slip to all the blobs of an active body in the function `active_body_slip` in the same file.

6. Run Monte Carlo simulations

We have a Markov Chain Monte Carlo code to generate equilibrium configurations. To use this code move to the directory `many_bodyMCMC` and inspect the input file `inputMCMC.dat`. The options are similar to the ones to run dynamic simulations, however, some options like the time step size are not necessary. We have only implemented the subroutines to compute the body-body interactions in *pycuda*, therefore, it is necessary to have a GPU with CUDA capabilities to use this code. To run a simulation use

```
python many_body_MCMC.py inputMCMC.dat
```

The output files are similar to the ones generated with dynamic simulations.

7. Software organization

- **body/**: it contains a class to handle a single rigid body.
- **boomerang/**: stochastic example, see documentation `doc/boomerang.txt`.
- **doc/**: documentation.
- **many_bodyMCMC/**: Monte Carlo code for rigid bodies.
- **mobility/**: it has functions to compute the blob mobility matrix $\mathbf{M}$ and the product $\mathbf{Mf}$.

- **multi_bodies/**: codes to run simulations of rigid bodies.
- **quaternion_integrator/**: it has a small class to handle quaternions and the schemes to integrate the equations of motion.
- **sphere/**: the folder contains an example to simulate a sphere whose center of mass is displaced from the geometric center (i.e., gravity generates a torque), sedimented near a no-slip wall in the presence of gravity, as described in Section IV.C in 1. Unlike the boomerang example this code does not use a rigid multiblob model of the sphere but rather uses the best known (semi)analytical approximations to the sphere mobility. See documentation `doc/boomerang.txt`.
- **stochastic_forcing/**: it contains functions to compute the product **M**ˆ{1/2}**z** necessary to perform Brownian simulations.
- **utils.py**: this file has some general functions that would be useful for general rigid bodies (mostly for analyzing and reading trajectory data and for logging).