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.

Note: We are still working on stochastic methods for suspensions of many rigid bodies. For now, the codes can do deterministic simulations for many bodies (see paper 2 below) but can only do Brownian Dynamics for a single rigid body (see paper 1 below).

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

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. Note that this does not include correction for blobs that overlap the wall, only for blobs that overlap each other. It is therefore important to keep blobs from overlapping the wall by a hard-core-like repulsion. Blobs can, however, overlap other blobs, and adding hard-core repulsion between blobs will introduce numerical stiffness.

1. Prepare the package

The codes are implemented in python 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 $\mathbf{M}$ and the matrix vector product $\mathbf{Mf}$. For some functions we provide a *C++* implementation which

can be around 5 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 for 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.

Donev: This is missing an explanation of active slip, maybe you added this recently after I pulled this.

Please don't forget to add it here also

Maybe you and Blaise can work together to put an input file here corresponding to the example from the paper

about a pair of dimers of active rods that spins

3. 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-fule inputfile.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`: 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 **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 **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).

- `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.

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).

4. Run dynamic 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 simulations for many bodies. In the future we will extend this code to allow for stochastic simulations of many bodies.

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

```
# 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 --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.
- `.time`: the wall-clock time elapsed during the simulation (in seconds).

List of options for the input file:

- `scheme`: Options: `deterministic_forward_euler_dense_algebra, deterministic_forward_euler, deterministic_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*` is second order accurate. The scheme `*dense_algebra` use 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++ and pycuda`. This option select the implementation to compute the matrix vector product **Mf**. See section 1 to use the C++ or pycuda implementation.
- `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**.

Donev: This is not urgent (and is not trivial) but since we may be doing a lot of stuff with spheres it would

be nice to add an option where the user specifies the *body-body* and body-wall force instead of the blob-blob

force. As you know specifying a force between blobs leads to loss of rotational invariance, and is also considerably

more expensive.

What I suggest is adding another option body_body_force_implementation (with None being one of the options)

and then one can specify one or the other and the forces are just added up in the end

As an example you can just do spheres interacting via Yukawa

- `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 Yukawa potential of the form (`U = eps * exp(-r / b) / r`) where `r` is the distance between blobs, `b` is the characteristic length and `eps` is the strength. This is the strength of the potential, `eps` in the above expression (see section 4.1 to modify blobs interactions).

- `debye_length`: (float) the blobs interact through a Yukawa potential (`U = eps * exp(-r / b) / r`), this is the characteristic length of the potential, `b` in the above expression (see section 4.1 to modify blobs interactions).

- `repulsion_strength_wall`: (float) the blobs interact with the wall with a Yukawa-like potential that imitates a hard core potential. The potential is (`U = eps * a * exp(-(h-a) / b) / (h - a)`) where `h` is the distance between the wall and the particle, 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 4.1 to modify blobs interactions). Note that the hard-core repulsion here is important since the Swan-Brady blob mobility is not well-behaved when the blobs overlap the wall.

- `debye_length_wall`: (float) the blobs interact with the wall with a Yukawa-like potential (`U = eps * a * exp(-(h-a) / b) / (h - a)`). This is the characteristic length of the Yukawa potential, `b` in the above expression (see section 4.1 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.

- `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.

4.1 Modify the codes

Right now, the slip on the rigid bodies and the interactions between blobs and between blobs and the wall are hard-coded in the codes. We explain here how the user can change these 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.

- active slip: the blobs can have an active slip (or swimming gait) as described in the Ref. 2. With this option 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 his ID (the name of the clones file without the path or the end `.clones`). The default slip is zero for all the blobs. The user can generalize this slip 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.

5. Software organization

- **body/**: it contains a class to handle a single rigid body.

- **boomerang/**: stochastic example, see documentation `doc/boomerang.txt`.

- **doc/**: documentation.

- **mobility/**: it has functions to compute the blob mobility matrix **M** and the product **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**^{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).