# RunAndTumble Library

Tomás Aquino

May 29, 2020

**Abstract**

These are brief notes on the structure of the RunAndTumble library code and how to use it to compile and run executable on a UNIX system.

## 1 Requirements

The RunAndTumble library is written in C++. The code base is header-only, and some test examples are provided. A compiler compatible with C++17, such as a sufficiently recent version of g++, is required to generate executables. Aside from the C++ standard template library, this library uses the nanoflann header-only library for kd-tree implementations (`https://github.com/jlblancoc/nanoflann`), and the Eigen header-only library for matrix algebra (`http://eigen.tuxfamily.org/index.php?title=Main_Page`). See also Section 3 for details on compilation. A simple processing script written in Matlab to make videos based on data from RunAndTumble simulations is found in the processing folder.

## 2 Code structure

The code is based on different structures dealing with different elements of the simulation. Model features must be chosen at compile time, and different model features are chosen with `using namespace` declarations inside main in src/RunAndTumble.cpp. Namespaces for the model, domain, particle initial conditions, and field initial conditions must be used. Available model setups are declared in Models.h, domain setups in Domains.h, and initial condition setups in InitialConditions.h, all found in include/Bacteria.

A brief review of the different elements follows.

## 2.1 Domain and grid

Some simulation elements rely on an underlying grid. We use a regular rectangular grid. Different nodes are identified as solid or void. Simple classes and utilities are provided to produce rectangular or spherical domains and solid inclusions.

## 2.2 Run-and-tumble dynamics

The run-and-tumble dynamics uses particle tracking, which is based on the CTRW and PTRW classes. The latter class is simply a wrapper for CTRW dynamics characterized by a timestep that is equal for all particles. A particle is characterized by a State object.

The details of the dynamics in each step are handled by a Transitions class. A transition in the run and tumble model has different elements:

- A Jump_generator object, which handles movement during the run phase.
- Two Orientation_generator objects, which handle angle reorientation during the tumble and wall-tumble phases, respectively.
- A StateSwitcher object, which handles changes of state between run, tumble, and wall-tumble.
- A Boundary object, which handles boundary conditions.

In the test model example:

- State: Composed of a vector position, an orientation angle, an integer tag numbering particles, and an integer state which takes the values 0 (run), 1 (tumble), and 2 (wall-tumble).
- Jump_generator: Jumps of fixed size at current orientation.

- Orientation_generator: Particles have a preferred nutrient value. The new orientation angle is gaussian-distributed with a given variance (cut off beyond $\pm\pi$), with a mean such that particles are aligned or anti-aligned with the local gradient depending on whether they are above or below the preferred value. If the local gradient is zero, the new angle is uniformly random.
- Orientation_generator_wall: Flips the current orientation angle.
- Boundary: Checks whether the closest grid element center to particle is a solid or a void. In the former case, the particle is placed at the center of the nearest void node in wall-tumble state.

Additional classes that may be used to replace some of these options are also provided, see in particular the header files in include/Stochastic/CTRW.

## 2.3 Scalar field dynamics

Scalar field dynamics, such as nutrient diffusion, are handled in an Eulerian framework using a Crank–Nicolson scheme. The necessary matrix algebra is handled using the Eigen library. The resulting matrix equation is solved using a sparse stabilized biconjugate gradient method, with an incomplete LUT preconditioner.

## 2.4 Coupling of particle and scalar field dynamics

Coupling of Eulerian and Lagrangian components, such as nutrient consumption by bacteria, is done by simple operator splitting, meaning at each time step, particle movement, field solution, and coupling dynamics are updated sequentially.

# 3 Compilation

Executables are compiled using the Makefile in the src folder. The Makefile uses the g++ compiler by default; if a different compiler is desired, the CC variable should be changed to its name. The path to the Eigen library is /usr/local/include/eigen3 by default. If a different path is desired, the PTHEIGEN variable should be changed as appropriate. Similarly, the path to the nanoflann header (nanoflann.hpp) is /usr/local/include/ by default, and if a different path is desired, modify the PTHKDTR variable.

A make.sh shell script for automatic compilation given a model choice is also provided in src. The shell script need to be given executing permission, which can be achieved by running

```
chmod +x make.sh
```

Each model is compiled into a dedicated executable. Running

```
./make.sh modelname domainname initialconditionparticles
   initialconditionfields
```

will produce the executable

```
RunAndTumble_
   modelname_domainname_initialconditionparticles_
   initialconditionfields
```

corresponding to the following declarations inside main in RunAndTumble.cpp:

```
using namespace model_modelname;
using namespace domain_domainname;
using namespace
   initial_condition_particles_initialconditionparticles;
using namespace
   initial_condition_fields_initialconditionfields;
```

Running the script make_tests.sh with no parameters builds all test models and places the executables in the tests folder.

## 4 Execution and parameters

The code is run from a command line by running the executable followed by parameter values, separated by spaces. The shell parameters to be passed to an executable are:

- Parameter set name: The parameter file identifier (see below)
- Run number: A nonnegative integer indexing the current run (for output file naming purposes only).

- Parameter directory [optional]: Directory to look for parameter files. Default: ../parameters
- Data directory [optional]: Directory for data output. Default: ../data

The remaining (dynamical) parameters are to be placed in dedicated files, in the appropriate directory according to the shell arguments above. Parameters are categorized by type, and should be placed in dedicated files. The types are:

- EulerianFields: Relating to the Eulerian fields.
- Domain: Relating to the domain.
- Discretization: Relating to the discretization.
- ICBacteria: Relating to the particle initial condition.
- ICEulerianFields: Relating to the Eulerian Fields initial condition.
- Output: Relating to output options.

If the parameter set name is, e.g., test_name, the naming convention is, e.g., parameters_Output_test_name.dat, for each parameter file. A list and brief description of the dynamical parameters of each type for a given executable can be obtained by running the latter without passing any shell parameters. Parameter files for the test examples are provided in the default folder.