BeadPack library manual

Tomás Aquino

January 21, 2021

Abstract

This is a brief manual describing the code structure, main processes implemented, and how to compile and use the code on a UNIX system.

1 Compilation and execution

The current library is written in C++ and header-only. Executables using the library must be compiled and linked to external dependencies locally using a compiler compatible with C++17. The code depends on the following external libraries:

- boost : C++ library implementing useful features and extensions (`boost.org`).
- CGAL : C++ interpolation library (`cgal.org`).
- Eigen : C++ matrix algebra library (`eigen.tuxfamily.org`).
- GMP : C arbitrary-precision arithmetic library (`gmplib.org`).
- MPFR : C multiple-precision arithmetic with correct rounding library (`mpfr.org`).
- nanoflann : C++ KDTree searching library (`github.com/jlblancoc/nanoflann`).

These should be obtained and setup according to their respective documentations before using the present library.

The following compilable example `.cpp` files are provided:

- `BeadPack.cpp` – Advective-diffusive particle tracking in 3d beadpacks with periodic boundary conditions.
- `BeadPack_Fp.cpp` – First passage times and distances to the fluid-solid interface for advective-diffusive particle tracking in 3d beadpacks with periodic boundary conditions.
- `BeadPack_Return.cpp` – Return times and distances to the fluid-solid interface for advective-diffusive particle tracking in 3d beadpacks with periodic boundary conditions.
- `BeadPack_Reactive.cpp` –Advective-diffusive particle tracking in 3d beadpacks with periodic boundary conditions and decay reaction at constant rate at the bead interfaces.
- `BeadPack_ReactionMap.cpp` – Fluid-phase mass consumed as a function of angle on bead surface for advective-diffusive particle tracking in 3d beadpacks with periodic boundary conditions and decay reaction at constant rate at the bead interfaces
- `BeadPack_Strips.cpp` – Track particle strips for fully-advective particle tracking in 3d beadpacks with periodic boundary conditions.
- `BeadPack_Statistics.cpp` – Statistics for 3d beadpacks with periodic boundary conditions.

The default examples are compiled for a body centered cubic with periodic boundary conditions on a (cubic) primitive cell of side 10^{-2}m. Support for periodic boundary conditions on non-cubic unit cells is also provided in the file `BeadPack_Model.h`. Different models can be compiled by changing the `using namespace` declaration at the top of the `.cpp` files.

By default, the provided `Makefile` for compiling the examples uses the gnu compiler g++. If you prefer to use a different compiler or have g++ under a different command name, edit `Makefile` to set the variable `CC` to the name of your compiler of choice. External library header files are set by the variable `PTHINC`, which is set to `/usr/local/include` by default. If you use a different location, edit this variable, or add additional paths to the `INC` variable. Exectuables must be linked to the GMP library. The corresponding path is set by the `PTHLIB` variable, which is set to `/usr/local/lib` by default and may also be changed if necessary. Alternatively, include and

library directories can be added to the local system PATH variable as usual. To compile the corresponding executables, open a command line, `cd` into the `src` directory, and run `make`. This should produce a final executable corresponding to each of the `.cpp` examples, named identically but without extension, in the `bin` folder. Each example may also be compiled separately. For example, `make BeadPack` compiles the `BeadPack.cpp` example.

An executable named `ExampleExecutable` is run from a command line by executing `./ExampleExecutable` followed by parameter values, separated by spaces. Running the example executables with no parameters prints a brief description and a list of the required parameters followed by short descriptions. Alternatively, parameter values may be placed in a plain text or csv file. For example, to run `ExampleExecutable` with the parameters in the file `ExampleParameters.dat` in the folder `parameters` folder, run

```
ExampleExecutable $(< ExampleParameters.dat)
```

2 Code structure

The basic structure of the code relies on a generic CTRW object, which handles a set of Particle objects. Each Particle object is associated with two State objects, describing the particle state associated with the next and previous turning points. The dynamics of these particles are then handled by a Transitions object. Upon a transition, Transition objects take the two particle states, modify the newer state according to the current transition, and set to older state to the previous newer state. While the generic CTRW object interface is designed to handle more general dynamics, for the particle tracking random walk (PTRW) dynamics considered here, each transition step is associated with a deterministic time step. The CTRW and Transition objects are then wrapped by a PTRW object which handles fixed-time-step transitions. At the completion of a given step, the two states of a particle then correspond to the current state and the state before the step. The PTRW object keeps track of system time.

For the present use, the Transition object may include any combination of advection, diffusion, and reaction processes. These are affected by the presence of the spherical (in two dimensions, circular) beads in a beadpack. The geometric information about the beadpack, such as bead centers and radii, is handled by a BeadPack object.

2.1 Transport processes

Regarding transport, the BeadPack library focuses on diffusion (with constant diffusion coefficient) and advection according to a heterogeneous flow field. The corresponding particle displacements are handled by a transport-related Transitions object for deterministic time steps. Within this object, particle displacements are handled by a JumpGenerator object, which produces a displacement given the current particle state. JumpGenerator objects implementing various methods for advective and diffusive steps are implemented. When both processes are present, a JumpGenerator is used to obtain the full displacement by summing the contributions from the respective JumpGenerators.

For diffusion, the following methods to compute displacement for a given time step are currently implemented:

- Stochastic forward-Euler temporal discretization of the Langevin diffusion equation.

For advection, if the flow field is known analytically, methods to use the analytical flow field as a function of position are available in the BeadPack library. However, the flow field must typically be obtained from separate flow simulations. The resulting flow field at an arbitrary set of points in the void space between beads is then to be provided by the user. The BeadPack library allows for computing advective displacements using interpolation to obtain the flow field at arbitrary particle positions. The following interpolation procedures are currently implemented:

- Two dimensions: Linear interpolation based on Delaunay triangulation and Sibson natural neighbor coordinates (using the CGAL library).
- Three dimensions: Linear interpolation based on Delaunay triangulation and natural neighbor coordinates (using the CGAL library).

For a given method to compute the flow field at an arbitrary location, the following methods to compute advective displacement for a given time step are implemented:

- Forward-Euler temporal discretization.
- Fourth-order Runge–Kutta temporal discretization.

Boundary conditions are handled by a Boundary object, which takes the previous and current state upon a transport transition, and enforces the boundary conditions on the current state. The following boundary conditions are currently implemented:

- Reflecting boundary conditions on bead surfaces.
- Periodic boundary conditions on a domain with faces perpendicular to the Cartesian axes.

The files `BeadPack.cpp`, `BeadPack_Fp.cpp`, and `BeadPack_Return.cpp` provide examples of how to make use of different combinations of these processes to set up conservative transport simulations. The file `BeadPack_Statiscs.cpp` provides examples on how to adapt the discretization time step for fixed-magnitude particle displacements using a dedicated Transitions object, which helps speeds up simulations by avoiding retention in low-velocity regions for many computational steps.

2.2 Reaction processes

Reaction processes may be added by using a Transitions object which combines a transport-related Transitions object as described above with a reaction, and applies them sequentially. The following reaction processes are currently implemented:

- Fluid-solid decay reaction at constant rate on bead surfaces.

The files `BeadPack_Reactive.cpp` and `BeadPack_ReactionMap.cpp` provide examples of how to combine transport and reaction processes.

2.3 Tracking collections of particles

In some applications, it is necessary to keep track not only of individual particles, but of different collections of particles and their relations to each other. This is addressed in the BeadPack library by the implementation of objects to keep track of ordered collections of particles handled by a CTRW object, and computing properties of such collections. Objects to handle sets of such collections are also available. The following collections are currently implemented:

- Strip: An ordered collection of particles, each connected to two neighbors. Can also handle closed strips (loops).

The file `BeadPack_Strips.cpp` provides an example of how to track strips of particles under advective transport.

2.4 Output

In the course of particle tracking simulations, quantities of interest relating to particle states can be computed, processed, and output. Typically, these quantities need to be measured a certain number times and/or when certain conditions are met, such as a given time being reached or a given control plane being crossed. The particle state quantities can be accessed for the current and old state of each particle through the CTRW or PTRW objects. A variety of helper StateGetter objects to obtain states from particles and quantities from states are provided. A number of generic Measurer objects, which compute and output different quantities across particles or particle collections, are also provided.

All the example `.cpp` files listed in Section 1 provide examples on how to measure different quantities under different conditions.