BCIM Documentation

Release 0.1

Dan Kolbman

May 28, 2015

1 Installation **3**
1.1 Intstalling Julia . . . 3
1.2 Python . . . 3
1.3 Matplotlib and Numpy . . . 3
1.4 BCIM . . . 3

2 Examples **5**
2.1 Running . . . 5
2.2 Running on a Cluster . . . 5
2.3 Quick Example . . . 5

3 Types **7**
3.1 Experiment . . . 7
3.2 Simulation . . . 7
3.3 Physical Constants . . . 8
3.4 Dimensionless Constants . . . 9
3.5 System . . . 10
3.6 Part . . . 10
3.7 Log . . . 10

Contents:

CHAPTER 1

Installation

BCIM's simulation portion is written in the Julia programming language. It is built using a relatively recent release of the development build (`0.4`). It may work on the current stable release (`0.3.6`), though it has not been tested.

1.1 Intstalling Julia

The nightly build is recommended as development on BCIM is done on the developmental release branch. Nightlies can be found on the Julia download page. Better yet, build julia from source using the directions on the Julia github.

1.2 Python

Post processing is done in `python 3.6`, though any release of `python 3` should work. Follow a guide online on how to install `python 3` for your environment.

1.3 Matplotlib and Numpy

BCIM uses `Matplotlib` for graphics and `Numpy` for numerical work. Both can be installed using `pip`:

```
pip install numpy matplotlib
```

1.4 BCIM

BCIM can be installed by cloning into the git repository on github:

```
git clone https://github.com/dankolbman/BCIM
cd BCIM
```

The `src` directory will have to be added to your shell path or the `src/julia/BCIM.jl` module can be inculeded by absolute reference inside your run files.

CHAPTER 2

Examples

2.1 Running

To run a simulation like the example below, Julia must be invoked from the top level directory of the repository (where the `src` folder resides), a from a directory that is appropriate for the `include` statement to find the source files. The simulation can then be run by passing the script to Julia:

> julia examples/num_part.jl

2.2 Running on a Cluster

Using slurm, many identical simulations can be run at once. The following will run the *adh.jl* script on 10 different cores with 2048Mb allocated on each.

> srun -N 10 -t 2000 –mem-per-cpu=2048 julia runs/vary_params/adh.jl

2.3 Quick Example

The following can be found in `examples/num_parts.jl` in the source. It creates an experiment with three trials and runs each one. It then modifies the parameters and creates a new experiment with a different number of particles. It repeats this three times for three different experiments each with three identical trials.

```
## Runs a experiments for diffrent numbers of particles
# Each experiment consists of three trials
# Saves data to data/numparts/ relative to run path

include("../src/julia/BCIM.jl")
#using BCIM

# Hack to allow asynchronous experiment runs
sleep(rand()*10)

# Our physical constants
pc = BCIM.PhysicalConst(  1.0e-5,            # dt
                          # Packing fraction
                          0.60,
                          # Eta
                          0.01,
                          # Temperature (K)
```

```
                                298.0,
                                # Boltzmann constant
                                1.38e-16,
                                # Propulsisions [ sp1, sp2 ]
                                [0.0,1.0e3],
                                # Repulsions [ sp1, sp2 ]
                                [1.5e4,1.5e3],
                                # Adhesions [ sp1, sp2, sp1-sp2 ]
                                [1.5e3, 0.0, 0.0 ],
                                # Cell division time ( 0 = no division)
                                [ 0.01, 0.01 ],
                                # Efective adhesive contact distance
                                0.1,
                                # Cell diameter
                                15.0e-4,
                                # Number of particles [ sp1, sp2 ]
                                [256,256])

##### 256 particles total
pc.npart = [128, 128]
# Initialize experiment with 3 trials in given directory with desired constants
exp = BCIM.Experiment("data/ex/256", 3, pc, false)

# Run the experiment
# Equilibriate for 1000 steps
# Collect every 1000 steps
# Run for 100000 steps
BCIM.run(exp, 1000:1000:100000)

##### Run again for 512 particles total
pc.npart = [256, 256]
exp = BCIM.Experiment("data/ex/512", 3, pc, false)
BCIM.run(exp, 1000:1000:100000)

##### 1024 particles total
pc.npart = [512, 512]
exp = BCIM.Experiment("data/ex/1024", 3, pc, false)
BCIM.run(exp, 1000:1000:100000)
```

CHAPTER 3

Types

3.1 Experiment

The experiment type is used to handle several trials of a simulation. It is responsible for creating paths for the trial simulations and saving parameters for them.

3.1.1 Functions

`Experiment` (*path*, *ntrials*, *pc*, *timestamp=false*)

Parameters

- **`path`** – the path to save the experiment directory in
- **`ntrials`** – nuber of identical simulations to run
- **`pc`** – the physical constants for the simulation systems
- **`timestamp`** – whether or not to append current time to the end of the experiment directory. Useful for avoiding name conflicts and over writing data.

Creates an experiment. Saves the pysical constants, `pc`, and dimensionless constants calculated from `pc`, to the `path` in `.dat` format. It creates `ntrials` number of simulations with parameters deteremined from `pc` and paths within the `path` directory. If `timestamp` is `true`, the date is append to the `path` name.

`run` (*exp*, *r*)

Parameters

- **`path`** – the experiment to run
- **`r`** – the step values to run each simulation for

Runs an experiment, `exp`, by invoking `run` on each trial simulation. `r` is the range to run each simulation and is of the format: `equilibrium:frequency:steps` where `equilibrium` is the number of steps to equilibriate the system for, `frequency` is how often to save the system state, and `steps` is how many steps to run for after equilibrium steps have been taken.

3.2 Simulation

The simulation type is used to contstruct a simulation system and run it for a desired amount of steps. It is responsible for steping the system and performing scheduled analysis of the system state, including writing the state to disk and calculating statistical quantities for the system.

3.2.1 Functions

Simulation (*dir*, *dc*, *log*)

Parameters

- **dir** – the directory path where a simulation directory will be created
- **dc** – Dimensionless constants to be used for the simulation
- **log** – The log to use for the simulation

Create a simulation. An `id` is assigned based on the next available directory in `dir` folling the convention: `dir/trial$id`. `dc` is a dimensionless contant object that is used to initialize the simulation system. `log` is a log object for the system to use for logging.

Simulation (*id*, *path*, *dc*, *log*)

Parameters

- **id** – a unique integer identifier for the simulation
- **path** – the directory where the simulation files will be stored
- **dc** – the dimensionless constants for the simulation
- **log** – the log to use for the simulation

Creates a simulation. `id` is a unique identifier for the simulation. `path` is where the simulation will place output files. `dc` is a dimensienless constant object for creating the system with. `log` is used to log simulation messages.

run (*sim*, *r*)

Parameters

- **sim** – the simulation to be run
- **r** – the step parameters to run for

Runs simulation, `sim`, for range `r`. `r` is of the format: `equilibrium:frequency:steps` where `equilibrium` is the number of steps to equilibriate the system for, `frequency` is how often to save the system state, and `steps` is how many steps to run for after equilibrium steps have been taken.

Example

```
# initialize sim for 100 steps, then run for 5000 steps
# and take measurements every 1000 steps
run( sim, 100:1000:5000 )
```

3.3 Physical Constants

The `PhysicalConst` type has many fields describing the physical (dimensional) parameters of the system:

PhysicalConst (*dt*, *phi*, *eta*, *temp*, *boltz*, *prop*, *rep*, *adh*, *contact*, *dia*, *npart*, *diff*, *rotdiff*)

Parameters

- **dt** – the time constant
- **phi** – the packing fration

- **eta** – the viscosity
- **temp** – the system temperature
- **boltz** – boltzmann's constant
- **prop** – the propulsion for each species
- **rep** – the repulsion for each species
- **adh** – the adhesion for each species
- **contact** – the contact distance as a fraction of the diameter
- **dia** – the diameter of each particle
- **npart** – the number of particles of each species
- **diff** – the diffusion
- **rotdiff** – the rotational diffusion

3.4 Dimensionless Constants

The DimensionlessConst type has many fields corresponding to dimensionless parameters of the system. A dimensionless type can be invoked by passing it a `PhysicalConst` type from which it will produce dimensionless parameters by scaling appropriatly.

DimensionlessConst (*dt*, *phi*, *eta*, *temp*, *boltz*, *prop*, *rep*, *adh*, *contact*, *dia*, *npart*, *diff*, *rotdiff*, *pretrad*, *prerotd*)

Parameters

- **dt** – the time constant
- **phi** – the packing fration
- **eta** – the viscosity
- **temp** – the system temperature
- **boltz** – boltzmann's constant
- **prop** – the propulsion for each species
- **rep** – the repulsion for each species
- **adh** – the adhesion for each species
- **contact** – the contact distance as a fraction of the diameter
- **dia** – the diameter of each particle
- **npart** – the number of particles of each species
- **diff** – the diffusion
- **rotdiff** – the rotational diffusion
- **pretrad** – the prefactor for translational diffusion
- **prerotd** – the prefactor for rotational diffusion

3.5 System

The `System` type is used to represent a physical system. It holds a list of particles which it is simulating, the dimensionless parameters of the system, and a `CellGrid` which is used to efficiently sort and simulate the particles.

`System`(*dc*)

Parameters **`dc`** – the dimensionless contstants for the system

Initializes a system using the dimensionless parameters `dc`. Constructs a cell grid and particles based on the specification of the parameters.

`uniformSphere`(*dc*)

Parameters **`dc`** – the dimensionless contstants for the system

Creates a list of particles, the number of which are specified by the npart field of `dc`, that have been randomly distributed in a sphere.

`step`(*s*)

Parameters **`s`** – the system to make a step on

Steps a system `s` by one step by calling the force calculation function.

`assignParts`(*s*)

Parameters **`s`** – the system to assign particles in

Assigns particles in a system into Cells in the system's `CellGrid`. Called by `Simulation` during a run periodically so collision checks can be made efficiently using the cell grid.

3.6 Part

The `Part` type is used to represent a particle in the system.

`Part`(*id*, *sp*, *pos*, *vel*, *ang*)

Parameters

- **`id`** – the particle id
- **`sp`** – the particle species
- **`pos`** – the position vector of the particle
- **`vel`** – the velocity vector of the particle
- **`ang`** – the angle vector of the particle

3.7 Log

`Log`(*path*, *verbose=false*)

Parameters

- **`path`** – the file to output logs to
- **`verbose`** – whether or not to pipe log to STDIN in addition to the file

`log`(*l*, *output*)

Parameters

- **`l`** – the log instance being logged to
- **`output`** – the output string to write

Index

A

assignParts() (built-in function), 10

D

DimensionlessConst() (built-in function), 9

E

Experiment() (built-in function), 7

L

Log() (built-in function), 10
log() (built-in function), 10

P

Part() (built-in function), 10
PhysicalConst() (built-in function), 8

R

run() (built-in function), 7, 8

S

Simulation() (built-in function), 8
step() (built-in function), 10
System() (built-in function), 10

U

uniformSphere() (built-in function), 10