SmolEmulate Documentation

for version 1.1

Steve Andrews

Contents

Chapter 1

Introduction

1.1 What is SmolEmulate?

SmolEmulate emulates the Smoldyn reaction algorithms using a 1-dimensional PDE simulations. These simulations represent rotationally symmetric 1D, 2D, and 3D diffusion-influenced reactions. SmolEmulate uses discrete time steps and fine spatial discretization to accurately represent continuous space.

The software uses text-based input from the user and outputs results as text and/or graphics, in the latter case by piping output to the gnuplot software. Overall, it is reasonably user-hostile. SmolEmulate opens with a main menu of options showing the different tasks that it can do. The user picks an option, SmolEmulate asks the user for parameters, then does the requested task, and returns to presenting the main menu again. It does not remember parameters from one round to the next.

This isn't finished code, but is really a framework that can be used for various tasks. Before doing one of those tasks, read through the code, or write more code, to make it actually work as desired.

1.2 Dependencies

The software is designed to work in conjunction with gnuplot, a free command-line plotting program. I installed gnuplot on my Mac using the MacPorts package manager, with "sudo port install gnuplot". If you use a different package manager though, then it's probably best to stick with that one rather than installing MacPorts. SmolEmulate should run without gnuplot, in which case it doesn't have any graphical output, but it also might not.

The only other dependencies are all in libSteve. SmolEmulate does not use OpenGL.

1.3 Models and algorithms

This software investigates several algorithms and compares them to a few models. The models are continuous-time models. They include the Smoluchowski model, in which molecules react immediately upon collision, the Collins and Kimball model, in which there is an "intrinsic reaction rate" for molecules that are in contact with each other, and the Doi model, in which there is a first order reaction rate for reactants that are within an interaction radius of each other.

The algorithms are the Andrews-Bray algorithm, which is a fixed time-step version of the Smoluchowski model, the $\lambda - \rho$ algorithm, which is a fixed-time step version of the Doi model, and the reactive hard sphere algorithm, which is a fixed-time step version of the Collins and Kimball model. Because these algorithms use fixed time steps, their simulations don't actually match the corresponding models, although they do approach the models in the limit of short time steps. See my file called Notes18C06 for details about the models and algorithms.

Chapter 2

Status as of 9/18/18

SmolEmulate is proving to be an essential tool for investigating the reaction rate functions. It has highlighted several problems, some dating back to my original 2004 algorithms and some in new algorithms. None of the problems in the old algorithms are significant for normal simulations, but could be issues in unusual situations. The problems in the new algorithms are mostly fairly small too, but again should be fixed. Here are the problems and probable solutions.

(1) For very short time steps, the numrxnrate and numrxnrateprob functions return results that are in error and that don't lead to smooth curves. Observe this by plotting reaction rate vs. step length, which is one of the SmolEmulate options. All looks good if the s' axis extends from 0 to 2. However, zooming in on the range from 0 to 0.01 shows some errors in that region. These are especially apparent for reversible reactions, with b' set to 1.1 or 1.01. I think the solution will be to remove the analytical extensions of the interpolation algorithms; I suspect that they are only valid if b' is substantially more than 1. Then, replace them with emulator calculations with s' down to e^{-4} and also to add a data point for $s' = 0$ (which is equal to 0). These should be relatively easy fixes and should address the problems well.

(2) For very small b' values and relatively short time steps, extrapolations are surprisingly jagged. I didn't expect that they would be accurate, but I was surprised that they weren't smooth. Observe this by plotting reaction rate vs. unbinding radius, choosing relatively small s' values (e.g. 0.2), which again is one of the SmolEmulate options. First of all, I don't know what is causing the jaggedness, which is something that I feel needs to be looked into. Secondly, I wonder if there's a better way of computing these extrapolations.

On thinking about this, I realized that there are two contributions to the reaction rate: the non-geminate molecules, which simply diffuse into the absorber at a constant rate, and a rate that is known from the irreversible reaction work. Also, the geminate molecules. These are produced at the unbinding radius at the constant rate from the non-geminate molecules, diffuse outward, some get sucked back in, they continue diffusing outward, and so on. It might make sense to only include the geminate molecules in the rdf and simply add a constant delta function source to it. Also, some analytical work could be helpful here. I have a poor sense of what the solution would be in the limit of short time steps, or even how to go about such a model (however, I'm fairly certain that the rdf would be equal to 1 for all $r > a$ because it's reversible and there's no net inward flux).

(3) Once the numrxnrate and numrxnrateprob functions are good for everything, then look at the bindingradius functions more carefully. The fitting procedures there seem to be remarkably good, but I'd like to do a bit more testing before I fully trust them for all parameter regions. For example, this testing could graph their results, much like I currently graph the numrxnrate functions in SmolEmulate.

(4) In Smoldyn itself, it seems that the lambda-rho algorithm is working reasonably well for many parameter options, but not for pgem, pgemmax, and pgemmaxw. Once all of the upstream stuff is fully debugged, then getting those to work correctly is the next step.

Chapter 3

Using SmolEmulate

3.1 Simulation parameters

Most of the SmolEmulate functions use the same simulation parameters, which are described here.

Algorithms and reaction probability. Algorithms are Andrews-Bray and Lambda-rho. They are identical except that reactants that are within a binding radius react with probability 1 for Andrews-Bray and with reaction probability P_λ for lambda-rho. Also, I have previously tabulated data and good interpolation functions for Andrews-Bray, but less good and newer results for lambda-rho.

Reversibility and binding and unbinding radii. The binding radius (σ_b) is the reactant separation at which a reaction does happen for Andrews-Bray and might happen for lambda-rho. If the reaction is reversible, then the unbinding radius (σ_u) is the product separation distance; if it's not reversible, then the unbinding radius is meaningless (and is set to -1 internally). Unbinding always happens to exactly this distance at the end of the time step. At steady-state, the rdf is 0 within the binding radius for Andrews-Bray. At steady-state, the rdf is 1 outside of the unbinding radius for both algorithms.

Rms step length (s). This is the mutual rms step length for the reactant molecules. It relates to the mutual diffusion coefficient and simulation time step as $s = \sqrt{2D\Delta t}$. In most of the emulator tasks, the diffusion coefficient and time step are not entered separately because the rms step length is sufficient.

Number of radial points in the rdf. The emulator rdf is tabulated with this many data points. It uses finite difference methods to compute PDE solutions. Data points are spaced with one exactly at 0, two that are very close on either side of the binding radius, and are otherwise equally spaced. If the reaction is irreversible, the emulator sets the maximum radius to about $10\sigma_b$. If the reaction is reversible with $\sigma_u > \sigma_b$, it sets the maximum radius to $\sigma_u + 4\sigma_b$. If the reaction is reversible with $\sigma_u < \sigma_b$, it sets the maximum radius to $5\sigma_b$. In all cases, the PDE solver exptrapolates the rdfs beyond their maximum radii.

Shape of the initial rdf. SmolEmulate supports well-mixed, Heaviside, and Smoluchowski initial rdfs. Well-mixed is everywhere 1. Heaviside is 0 inside of some radius (which the user has to enter) and 1 outside of that radius. It can be used to create a well-mixed system that went through one iteration of absorption with the Andrews-Bray algorithm. Smoluchowski follows the Smoluchowski model; for example, if it is irreversible, then the rdf is $1 - \frac{\sigma}{r}$, where σ is the binding radius.

Level of precision (ϵ). Most algorithms run until the fractional change in reactive flux from one iteration to the next is less than the level of precision entered here. In general, using 1e-4 is minimal, 1e-5 is good, and 1e-6 is very good, although larger or smaller numbers may be better depending on the circumstance.

3.2 Computing simulated reaction rate from parameters with graphics

The user is prompted to enter simulation parameters. Then, the emulator runs and displays the rdf as it goes along. At the end, it prints out the emulator results and compares them with the results from the functions that use tabulated data.

3.3 Compute simulated reaction rate from pre-selected parameters

This is identical to the prior task, except that the user isn't prompted to enter simulation parameters. Instead, they are hard-coded into the software. This task is best used for debugging issues, where the same simulation is wanted over and over again, and it's a nuisance to enter the same parameters over and over again.

3.4 Run Andrews-Bray calibration

This scans through the simulation parameters and runs the emulator for each parameter value. Results are printed out to the standard output for copying and pasting in the relevant table lookup and interpolation function.

3.5 Run lambda-rho calibration

As above, this scans through the simulation parameter and runs the emulator for each parameter value. Results are printed out to the standard output for copying and pasting into the relevant table lookup and interpolation function.

3.6 Compute time-dependent reaction rate

This computes the reactive flux in the simulation as a function of the number of time steps. Results are displayed to the gnuplot graphics window.

3.7 Graph numerical reaction rate

This graphs the numerical reaction rate constant on the gnuplot output for both the lambda-rho and Andrews-Bray algorithms, taking data from the lookup functions. Graphs are shown form P_λ equal to 0.1, 0.2, ..., 0.9, 1.0.

3.8 Compute steady state flux without graphics

This is identical to the first task, which was to "compute simulated reaction rate from parameters with graphics", except that no graphics are used here. This makes the code run a little faster and is more convenient if you already know what the graph looks like anyhow.

3.9 Rdfs for graphical output

The radial distribution functions plotted are worth a little thought.

In Smoldyn itself, the algorithm is

diff abs unbind **obs** diff abs unbind **obs** diff abs unbind **obs**

where diff is diffusion, abs is absorption, unbind is unbinding, and obs is observation. Diffusion makes the rdf smooth, absorption creates a step at the binding radius and 0 concentration inside if it's the Andrews-Bray algorithm. If the reaction is irreversible, then this is the observed rdf. If the reaction is reversible, then there is unbinding first. It is to a fixed unbinding radius, so the observed rdf is the same as before but also has a delta function at the unbinding radius. If the unbinding radius is at or inside of the binding radius, then the rdf has a negative slope at the binding radius and continues to be below 1 for a little ways outside of

this. This is okay because it shows a net influx of those molecules, but they are exactly offset by the net eflux from the delta function.

The emulator uses a slightly different ordering because it's hard to work with delta functions numerically. In the emulator, each unbinding is diffused separately from the rest of the rdf and then added to it after the diffusion step. Observation is then after the absorption step. The result is

diff unbind abs **obs** diff unbind abs **obs** diff unbind abs **obs**

Here, unbinding is of a Gaussian (a 3D Gaussian, which isn't exactly a Gaussian in 1D). Note that the unbinding shown here is always from the prior time step. Thus, if the observed rdf were to be made to be the same as the Smoldyn one, then it would the same as this, plus a delta function at the unbinding radius.

From this analysis, I'm pretty sure that the emulator output is correct, despite the fact that it's a little less than 1 for a small region outside of the unbinding radius.

3.10 Current status

Everything seems to work pretty well at present, for 3D reactions. I haven't done much with 1D or 2D yet. However, there are some concerns for 3D reactions. Looking at the graphs of reduced reaction rate vs. reduced rms step length, there are substantial errors for small s' values, especially when σ'_u is small, meaning between 0 and 1.5, or so. I don't know if this arises from inaccuracies in the lookup tables, too small lookup tables, and/or lack of exact functions for small values.

Chapter 4

Code

4.1 Data structures

4.1.1 emu

```
typedef struct emustruct {
  int dim;               // system dimensionality: 1,2,3
  char algorithm;        // algorithm: a,l,h
  int reversible;        // reversible or not: 0,1
  double difc;           // diffusion coefficient
  double dt;             // simulation time step
  double rmsstep;        // diffusive step size
  double bindrad;        // binding radius
  double unbindrad;      // unbinding radius or -1
  double rxnprob;        // reaction probability or -1
  double epsilon;        // precision level
  char initialrdf;       // initial rdf shape: w,h,s
  double initialrdfval; // value for initial rdf
  int npts;              // points in RDF
  double dr;             // spatial step size
  double *r;             // vector of radial positions
  int nrdf;              // number of RDFs (=2)
  double **rdf;          // RDFs
```

The `emustruct`, which stands for emulator, is the basic structure of the emulator.

- `dim` is the system dimensionality, which can be 1, 2, or 3.
- `algorithm` is the reaction algorithm being used. Options are: 'a' for Andrews-Bray (which is the default), in which the forward reaction always occurs for molecules that are within a binding radius of each other; 'l' for λ-ρ, in which molecules that are within the binding radius of each other react with some fixed probability and are left alone if they don't react; and (h) for reactive hard sphere, in which molecules within the binding radius of each other react with some fixed probability and are reflected off of each other if they don't react.
- `reversible` is 1 if the reaction is assumed to be reversible and 0 if not.
- `difc` is the mutual diffusion coefficient (D in equations). It is isotropic in all dimensions. For convenience, one can assume that there is one A molecule at the origin and lots of B molecules that diffuse with diffusion coefficient `difc`.
- `dt` is the simulation time step (Δt in equations).
- `rmsstep` is the root mean square diffusive step size (s in equations). It is set equal to $\sqrt{2D\Delta t}$

- `bindrad` is the binding radius (σ_b in equations).
- `unbindrad` is the unbinding radius (σ_u in equations). This is only relevant for reversible reactions. The application of this unbinding radius may be algorithm dependent.
- `rxnprob` is the probability that a reaction occurs for two particles that are within a binding radius of each other at the end of a time step.
- `epsilon` is the precision for the iterative algorithms. They run until the fractional difference between the results from successive steps is less than `epsilon`. This value is initialized to -1 and has a default value of 0.0001.
- `initialrdf` and `initialrdfvalue` specifiy the shape of the initial emulator rdf. Options for the character are: 'w' for well-mixed, 'h' for Heaviside function, and 's' for Smoluchowski. Only the Heaviside function requires that the value be set, which is the radius at which the initial rdf changes from being 0 within it to being 1 outside of it.
- `npts` is the number of spatial points.
- `dr` is the spatial step size.
- `r` is a vector of radial positions (r_i in equations).
- `nrdf` is the number of rdf vectors. It's always set to 2.
- `rdf` is the list of RDFs. `rdf[0]` is the radial distribution function after the absorption step has happened and called `rdfa` in many functions. `rdf[1]` is the radial distribiution function after the diffusion step has happened and called `rdfd` in many functions.

4.2 Functions

4.2.1 Memory management

`emuptr emualloc(emuptr emu,int npts);`
Allocates and initializes an emu structure and returns a pointer to it. This can also resize an existing structure. `npts` is the desired `emu->npts` value, for the number of radial values. This allocates the `emu->r` and `emu->rdf` vectors. Enter `emu` as `NULL` for allocating the data structure and as a pointer to an existing data structure to replace the three radial vectors with different size ones. Returns `NULL` for failure to allocate memory.

`void emufree(emuptr emu);` Frees an emu pointer and all of its contents.

4.2.2 Data structure output

`void emuhelp();`
Displays help stuff.

`void emuoutput(emuptr emu);`
Outputs all of the settable contents of the emu structure to the standard output.

`int emucheckparams(emuptr emu);`
Checks most of the parameters of an emu structure to ensure that they are within reasonable limits, sending output to the standard output. Prints out the number of warnings and errors at the end. Returns the number of errors.

4.2.3 Structure set up

`int emusetparam(emuptr emu,char *param,double value);`

Sets the value of the emu structure parameter named `param` to value `value`. This does not check values for being reasonable and nor does it update parameters other than the selected ones to preserve consistency (e.g. changing "difc" does not result in "rmsstep" being updated and vice versa). For parameters that are not doubles, including "algorithm" and "npts", simply cast the desired parameter value to a double and this will cast it back internally. If "npts" is changed, this function erases any existing information in the radial vectors.

Returns 0 for success, -1 for failure to allocate memory (only relevant for "npts"), or -2 for unknown parameter name. If an error code is generated, this also prints the error to standard error. Ignoring this return value is acceptable if the standard error will be visible.

`int emuhardcodeparams(emuptr emu);`

This is probably a poor function for the long term but is here and convenient for now. It hard codes a set of emu structure parameters to a set of specific values.

`int emuinputparams(emuptr emu);`

Inputs the emu structure parameters one at a time from the user.

4.2.4 Structure set up

`int emusetrmsstep(emuptr emu);`

Sets the `rmsstep`, `difc`, and/or `dt` parameters using the equation $s = \sqrt{2D\Delta t}$. This uses whatever information is available to compute the others, and sets values to 1 if information isn't available (i.e. the value is negative). Returns 0.

`int emumakervect(emuptr emu);`

Sets the `emu-r` vector with the radial positions and the `emu->dr` value with the radial step size, based on values in the `bindrad`, `unbindrad`, `reversible`, and `npts` elements. This sets the maximum radius to $10\sigma_b$ if irreversible, to $\sigma_u + 4\sigma_b$ if reversible and $\sigma_u > \sigma_b$, and to $5\sigma_b$ if reversible and $\sigma_u < \sigma_b$. It computes the spacing, Δr, based on this maximum radius and the number of points. In all cases, $r_0 = 0$, there is a radius point at $0.9999\sigma_b$, another point at $1.0001\sigma_b$, and all other radius points are separated by the computed Δr value.

`int emumakerdfvect(emuptr emu,char type);`

Initializes both rdf vector values, `emu->rdfa` and `emu->rdfd` for a given function type. Normally, this should be called with `type` equal to 'd' for default, meaning that this function should look in `emu->initialrdf` for the correct initial rdf shape. However, if a different initial shape is desired, it can be entered here. The `r` vector should already be set up, using `emumakervect`. Output is:

`type`	reversible	rdf for $r < \sigma_b$	rdf for $\sigma_b < r < \sigma_u$	rdf for $\sigma_u < r$
w (well-mixed)	either	1	1	1
h (Heaviside)	either		0 for $r < v$ and 1 for $r \geq v$	
s (Smoluchowski3D)	no	0	$1 - \frac{\sigma_b}{r}$	N/A
s (Smoluchowski3D)	yes, $\sigma_u > \sigma_b$	0	$1 - \frac{\sigma_u - r}{r(\sigma_u - \sigma_b)}$	1

4.2.5 Top level functions

`int emuactivationlimit(emuptr emu);`

Computes reactive flux for a single step, starting from well-mixed state. Displays results to standard out. This is somewhat obsolete because I now feel that the activation-limited reaction rate is better defined from the Noyes relation ($k^{-1} = k_{diff.}^{-1} + k_{act.}^{-1}$) rather than from the initial reaction rate from a well-mixed state.

`int emustepbystep(emuptr emu);`

Runs diffusion and absorption algorithms over many steps, displaying rdfs to gnuplot as it goes along. This function displays final results to standard out and compares them with interpolations from tabulated results, from the rxnparam.c library. It stops when the reactive flux has reached steady-state, as defined by its fractional change from one step to the next changing by less than ϵ.

`int emusteadystate(emuptr emu);`

Runs diffusion and absorption algorithms to steady state, again defined as the reactive flux having a fractional change less than ϵ between time steps. This does not display results to graphics, but does output final results and compares them to interpolations from tabulated results from rxnparam.c. The text output is identical to that from `emustepbystep`.

`int emurateconstant(emuptr emu);`

Runs the simulation with the given parameters and displays the reactive flux to the gnuplot graphics window as it goes along to show the time-dependent reaction rate constant. This also displays the reactive flux to the standard output. At the end, it computes various results from the final flux value, which are likely to be somewhat close to steady-state, and displays them.

`int emugraphnumrxnrate(emuptr emu);`

Graphs the numerical reaction rate constant showing the reduced rms step size on the x-axis and the reduced reaction rate constant ($\frac{k\Delta t}{\sigma_b^2}$) on the y-axis. All results are from interpolations of tabulated results using the numrxnrateprob function, not from new emulator computations. The lines are for probabilities of 0, 0.1, 0.2, ..., 0.9, 1.0.

4.2.6 main function

`int main(void);`

Main program entry point. Displays the main menu, gets input from the user, and then does a few things for each task. This primarily involves getting the necessary parameters, setting up rdfs or other data structures, and then running the appropriate top level function.