Documentation for SmolCrowd

Steve Andrews, May 2007
updated: March, 2011, November, 2014, June 2019

Introduction

This is a simple program that creates Smoldyn-readable files that describe crowding objects. These files are not complete Smoldyn configuration files, but only list the crowding objects. All information about the objects to be created is gotten from the user through a series of questions.

Running SmolCrowd

Upon running, the program asks the user for a few things:

- System dimensionality. Enter a number between 1 and 3.
- Low and high coordinates on each dimension. These values define the boundaries of the system, all of which are always periodic. Sometimes, I use 0 for the low coordinate and 100 for the high coordinate. Other times I go for -25 to 25.
- Desired crowding fraction (phi). This is the fraction of the system volume that's occupied. It's also the occupied area fraction for 2D and the occupied length fraction for 1D. Phi can be as low as 0 and, conceptually, as high as 1. Realistically, phi can't exceed about 0.35 for 3D and 0.6 for 2D.
- Crowder radius, or minimum if a range is desired (rmin).
- Enter 0 for all crowders the same size or power law slope (gamma). To get a power-law distribution of sphere radii, enter the power-law slope. This slope should be negative; typically between -2 and -4 work well. The actual power law slope, which SmolCrowd will determine and output, is typically different from this value due to sphere packing constraints.
- Extra radius, which can overlap and is not part of phi. This radius is simply added to the radius of each sphere but is not included in any calculations or in determining overlap. The idea is that if a tracer molecule has radius r_t and a crowder has radius r_c, then it's equivalent to say that the tracer has radius 0 and the crowder has radius r_c+r_t. In this case, r_t can be entered as the extra radius.
- Smallest permitted crowder edge distance from the origin. If you want to exclude a sphere around the origin from having any overlap with crowders, then enter that radius.
- Largest permitted crowder edge distance from the origin. If you want to only have crowders out to a certain radius, then enter that radius.
- File name to which the data will be saved.
- If you want a compartment for the crowders. Smoldyn compartments are composed of surfaces, which may be disjoint, and interior-defining points. If you say yes, then this defines a compartment using the sphere surfaces as the compartment surface and the

center of each sphere as the interior-defining points. Note that if two spheres partially overlap, then the overlap region is not within the compartment.

- If you want a molecule in the middle of each sphere and, if so, whether to include spheres that have their centers outside of the system volume and what the molecule species name should be. One reason why you might want to put a molecule inside each sphere is for computing radial distribution functions of other molecules about spheres. Currently, Smoldyn only computes radial distribution functions between pairs of molecules, so this enables radial distribution functions to be computed with spheres as well.
- If you want double spheres that are concentric with the originals and, if so, what their total radius should be (this ignores the extra radius parameter entered previously). If you say yes, then this defines a second complete set of spheres that are concentric with the originals, but that have a different radius. This is useful if two different species should reflect off of the spheres at different radii. Note that both sphere outputs output the Smoldyn statement "action all(all) both reflect", meaning that both the original and double spheres will reflect all molecules. This is probably not desired, so the reflection rules need to be corrected elsewhere.

Runtime varies tremendously. It's typically only a few seconds for low crowding fractions but then increases very rapidly toward infinity once it becomes hard to fit all of the spheres.

I haven't yet observed double or triple wrapped spheres, although I don't see why they shouldn't work. Also, be forewarned that the power-law slope of the power-law distributed spheres likely won't be what is asked for due to the way the packing algorithms work. What is needed is either a new function within this program to measure the actual power-law slope, or a tool in Excel or something else that can measure it.

SmolCrowd code

Data structures

No data structures are declared here. However, a data structure convention is assumed by several functions.

`double *spheres`

This is a list of spheres (or circles or line segments, all of which are considered to be spheres here) with 4 elements of the array for each sphere. The first elements give the center location and the next one is the radius:

`spheres[4*j+d]`	center location of sphere *j* on dimension *d*
`spheres[4*j+dim]`	radius of sphere *j*

If a radius is listed as negative, it means that that sphere has been killed. Only 1, 2, and 3-dimensional spheres are considered in this program.

Functions

`int Sphereintsct(int i,int j,double *spheres,int dim,double radextra);`

Returns 1 if spheres `i` and `j` intersect and 0 if not. `spheres` is the list of spheres and `dim` is the system dimensionality. 0 is returned if `i` equals `j` (*i.e.* a sphere cannot intersect itself) or if either radius is negative (*i.e.* killed spheres aren't considered). `radextra` is an extra radius that is added to both sphere radii.

`double measurephi(double *low,double *high,double *spheres,int n,int dim,int itmax,double originradmin,double originradmax);`

Measures actual sphere volume density in the system with a random sampling approach. As usual, `low` and `high` are the boundaries of the system, `spheres` is the list of spheres, `n` is the number of spheres in the list, and `dim` is the dimensionality of the system. `itmax` is the numer of random points that are sampled; a value of 10000 is often good. For efficiency reasons, this does not treat killed spheres properly, so make sure that the spheres list is compacted first and that all `n` spheres are good. This returns the volume coverage, which is between 0 and 1, inclusive. The method used here is that points are sampled randomly; all points that are in any sphere are counted, and the ratio of counts to total samples (`itmax`) is returned. If this function is run many times, the average of the returned values will equal the actual phi and the standard deviation of the returned values will equal $[\texttt{itmax}*\phi*(1-\phi)]^{1/2}$, which is roughly equal to the error returned from one run.

`double calcphi(double *low,double *high,double *spheres,int n,int dim,double originradmin,double originradmax);`

Calculates sphere volume density in the system, assuming that no spheres overlap each other. Negative radius spheres are excluded from the calculation. Also, any spheres that overlap a high side of the system volume are excluded so as to not overcount spheres that are wrapped with periodic boundaries.

`int makeradhist(double *spheres,int n,int dim,double factor,double **histptr,double **scaleptr)`

Allocates and sets up memory for a histogram of sphere radii, used to analyze the radius distribution. `spheres` is a list of spheres, which must be previously compacted, `n` is the number of spheres, and `dim` is the dimensionality. The histogram is made to have log spacing, spaced so that the smallest spheres is in bin number 1 and the largest is in the next to last bin. For each bin, the minimum radius is `factor` (which needs to be >1) times the minimum radius of the prior bin; also, bin boundaries are made to be at integer multiples of `factor`. Send in histptr and scaleptr as pointers to the hist and scale variables. The function returns 0 for failure and otherwise it returns the total number of histogram bins. See the histogram functions in RnSort.c.

`void freeradhist(double *hist,double *scale);`

Frees a histogram that was set up by `makeradhist`.

`void fillradhist(double *spheres,int n,int dim,double *hist,double *scale,int hn);`

Using a histogram already set up with `makeradhist`, this goes down the list of spheres and fills in the histogram. `spheres` must be already compacted, `n` is the number of spheres, `dim` is the dimensionality, `hist` and `scale` are the vectors that were set up with `makeradhist`, and `hn` is the number of histogram bins which was returned from `makeradhist`.

`int showradhist(double *spheres,int n,int dim);`

Creates, fills in, displays, and frees a histogram for the sphere radius distribution. This asks the user for the bin size factor to use. Returns 1 for done, 0 for continue.

`int countclusters(double *spheres,int n,int dim,double radextra)`

This function is supposed to count the number of clusters, where a cluster is defined as one or more crowders that are connected to each other through their extra radii overlapping each other but that are disconnected from other crowders. I don't know why it doesn't work, but this function doesn't seem to account for all overlaps, and so claims that there are more disjoint clusters than there really are. I tried to debug it for a while but then gave up.

`void writespheres(double*spheres,int n,int dim,char *comment);`

Writes all spheres with non-negative radius, in `spheres` and of length `n`, to a Smoldyn-readable file. The file name is gotten from the user. An optional comment line is added to the file, near the top. The surface name is always "spheres."

`int makespheres(int nnew,double rmin,double gamma,double *low,double *high,double *spheres,int n,int dim);`

Adds `nnew` `dim` dimensional random spheres to a list of `n` existing spheres. Their centers are uniformly randomly distributed between `low` and `high`, which are `dim` dimensional vectors. Set *gamma* to zero for all spheres to have radius `rmin`; otherwise set *gamma* to a value less than -1 for a power law distribution of radii with power *gamma* and minimum radius `rmin`. The function returns the total number of spheres in the list, which is `n+nnew`. Any overlaps between spheres are ignored. It is assumed that the array has been allocated large enough.

`int compactspheres(double *spheres,int n,int dim);`

Compacts a list of spheres, pushing all negative radius ones to the high index end of the list. The center positions of the negative radius ones are not preserved. The order of positive radius spheres within the list is preserved. Returns the number of remaining spheres.

`int wraplastsphere(double *low,double *high,double *spheres,int n,int nmax,int dim);`

Wraps just the last sphere, which is sphere number `n-1`, on all dimensions. As usual, low and high define the edges of space, spheres is the list of spheres, n is the number of spheres that are defined, nmax is the allocated size of spheres, and dim is

the system dimensionality. This returns the new total number of spheres. No intersections are checked. Spheres should wrap properly on as many dimensions as needed, and wrap (up to once each) in both directions if they overlap both sides of space in any dimension.

`int wrapspheres(int d,double *low,double *high,double *spheres,int n,int nmax,int dim);`

Adds additional spheres to the list to account for spheres that overlap the edges of periodic boundaries, only on dimension `d`. `n` is the number of spheres in the list, `nmax` is the total allocated space, and `d` is the dimension to be wrapped. `low` and `high` are `dim` dimensionsional vectors for the low and high corners of space. Returns the new total list length. Killed spheres, which have a negative radius, are ignored by this function. To wrap the system on multiple axes, just call this function for each axis sequentially; multiple wraps for a single sphere will be accounted for automatically.

`int unwrapsphere(int i,double *low,double *high,double *spheres,int n,int dim);`

For this situation in which sphere `i` is to be killed, this is used to kill off all wrapped images of sphere `i` and returns the number of spheres that were killed. This needs the radius of sphere `i`, so make sure that this function is called before sphere `i` is killed; this does not kill sphere `i`. As usual, `low` and `high` are the boundaries of space on each dimension, `spheres` is the list of spheres, `n` is the number of spheres in the list, and `dim` is the system dimensionality. This properly accounts for singly, doubly, and triply wrapped image spheres.

`int MakeSph2Phi(double phi,double rmin,double gamma,double *low,double *high,double *spheres,int n,int nmax,int dim,double originradmin,double originradmax);`

Adds new spheres to `spheres`, which already had `n` spheres in it, until the volume density equals `phi`, if possible. If `gamma` is 0, the spheres that are added all have radius `rmin`; otherwise a power-law distribution with slope `gamma` and minimum radius `rmin` is used, in which case `gamma` should be significantly less than –1. `phi` is the desired volume density, which is a number between 0 and 1. The system dimensions on each coordinate are defined by `low` and `high`. `nmax` is the allocated size of `spheres` and `dim` is the dimensionality of space. This adds spheres randomly one at a time; after each is added, it is wrapped as appropriate and then checked for intersections with existing spheres. If there is an intersection, the new sphere and any images are removed and another try is made. After many failures, a random sphere (biased for small spheres) is removed, and the cycle repeats. The function returns to the user either once the desired density is first exceeded or when the density cannot be achieved even after many removals. There is no certainty that the final power law slope will be anywhere close to the requested value.