RKP_Suite Documentation

Usage Guide and Reference Manual

Nathaniel Morrison

9/6/2020

This work was partially supported by grants from the National Science Foundation DMS–1821254.

Table of Contents

Introduction

The RKP_Suite is a set of programs designed to simulate kinetoplasts, particularly as they are chemically dissolved. Kinetoplasts are sheets of many interconnected circular loops of DNA found primarily in protists of the class *kinetoplastea*. The chainmail-like structure must be dissolved and reformed when these microbes reproduce, and studying the mechanisms by which this happens may allow the process to be disrupted in pathogenic members of *kinetoplastea*, including *Trypanosoma brucei*, the causative agent of sleeping sickness. Further information on kinetoplasts can be found on Wikipedia[1] or, if that's not peer-reviewed enough for you, in articles like this one[2].

Kinetoplasts are modeled as networks of nodes with fixed spatial coordinates linked by edges. Dissolution, such as must occur during reproduction, is modeled as periodic removal of random nodes from this network. Multiple types of kinetoplast structure may be simulated, and the user may change a large number of parameters governing the generation of such models, both from the front end through the kinetoplast, hkinetoplast, tkinetoplast, rkinetoplast, kinetoplast3D, and hkinetoplast3D classes and from the back end by manually changing constants within the code.

This documentation is intended to provide a general overview of how kinetoplast models are built and simulated and describe the basic functionality of each module, class, and script in the RKP_Suite. It is not a complete reference guide detailing the input and output of every function, because ain't nobody got time for that. See the inline comments in the code for more detailed information on how algorithms are actually implemented, and for a terrifying look into my stream of consciousness.

The remainder of the introduction will detail the basic software requirements necessary to run the RKP_Suite. The Simulation Methods section will then detail the various types of model that can be generated and the procedure used to simulate chemical dissolution, called a "dissolution search". The Data Analysis Methods section will describe the various images and datafiles that can be generated during a simulation to record its behavior. The General Usage section will detail the functionality of each method in the kinetoplast, hkinetoplast, kinetoplast3D, and hkinetoplast3D classes, as well as how to use the various scripts that automatically run dissolution searches and interpret the results. The rkinetoplast and tkinetoplast modules are not discussed, as they work (almost) identically to hkinetoplast, having (almost) all the same methods and very similar attributes. Finally, the Known Limitations section will discuss the copious number of known limitations, bugs, and ways to make the programs crash horrifically.

Requirements to Run

1. A computer. Bet you didn't see that coming.
2. A Python 3 installation.

[1] https://en.wikipedia.org/wiki/Kinetoplast

[2] Balaña-Fouce, Rafael ; Álvarez-Velilla, Raquel ; Fernández-Prada, Christopher ; García-Estrada, Carlos ; Reguera, Rosa M. *International Journal for Parasitology: Drugs and Drug Resistance*, 2014-12, Vol.4 (3), p.326-337

3. A Microsoft Excel installation. Excel 2010 or later is recommended. A compatible program, for example LibreOffice Calc, should also work.
4. The following Python modules should be installed, if they are not already:
    a. numpy
    b. scipy
    c. matplotlib
    d. networkx
    e. xlsxwriter
    f. xlrd
5. The following files should be placed in the same folder:
    a. bf_search.py
    b. collection_plot.py
    c. delta_p_max_plot.py
    d. delta_p_max_plot_avg.py
    e. dissolution_plot.py
    f. dissolution_plot_avg.py
    g. graph_funcs.py
    h. grid_gen.py
    i. hkinetoplast.py
    j. kinetoplast.py
    k. rkinetoplast.py
    l. tkinetoplast.py
    m. prob_funcs.py
    n. search_conductor.py
6. The following files should be placed in the same folder, different from the one in step 5:
    a. bf_search3D.py
    b. dissolution_plot3D.py
    c. graph_funcs3D.py
    d. grid_gen3D.py
    e. hkinetoplast3D.py
    f. kinetoplast3D.py
    g. prob_funcs3D.py
    h. search_conductor3D.py

Simulation Methods

Types of Model

The package supports four broad types of structure: hexagonal, triangular, rectangular, and randomized. The hexagonal and randomized models can be two- or three-dimensional; the triangular and rectangular are exclusively 2D. Below are descriptions of the structure of each broad category of model, and the parameters which can be altered. Details on how the generation algorithm is implemented can be found in the inline comments within the code itself.

Randomized2D

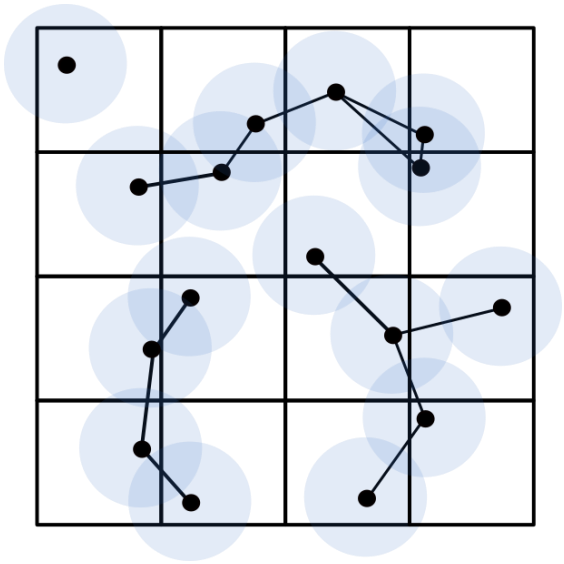

Figure 1.1: Basic randomized2D grid, with *n*=*m*=4 and probability field radius approximately 0.5. The nodes (black circles) are randomly placed within their gridboxes, and the probability fields are the light blue circles surrounding them. This example assumes a link probability of 1 for any pair of nodes with overlapping probability fields.

The randomized2D grid is a lattice of semi-random nodes within a hidden, underlying rectangular grid. The space is divided into an *n* by *m* grid of squares of side length 1, where *n* is the number of columns and *m* the number of rows. Within each square, exactly one node is randomly placed, surrounded by a "probability field" of user-defined radius. See Fig 1.1 for an example.

Once the grid is generated, the program will generate connections between nodes. Any two nodes with overlapping probability fields are eligible for a connection, with the probability of a link dependent upon the two nodes' probability fields. Call a hypothetical pair of eligible nodes node1 and node2, with probability fields pf1 and pf2. The user is able to manually enter a probability function f(r,θ) in the prob_funcs.py file, which may be both radially and angularly dependent (though there are significant limitations to angular dependence; see the Known Limitations section) defining the probability field within a circle of the user-defined radius. Currently, the program integrates pf1 over the overlap region

(where the two probability fields intersect) and divides by the area of pf1 to give the link probability. A random number between 0 and 1 is then generated. If the number is less than or equal to the link probability, an edge is created joining the two nodes. Otherwise, no edge is created.

You can, however, create practically any link probability formula with just very simple changes to prob_funcs.py. Previously, for example, I integrated pf1 over the overlap region, then I integrated pf2 over the overlap region, then I multiplied the results to get the link probability. The code for this still exists as a comment in prob_funcs.py. If you want a (practically) guaranteed edge between any two nodes with intersecting probability fields, add a very large scalar to the probability function. I've been using f(r,θ)=10000, and that seems to guarantee connections very well.

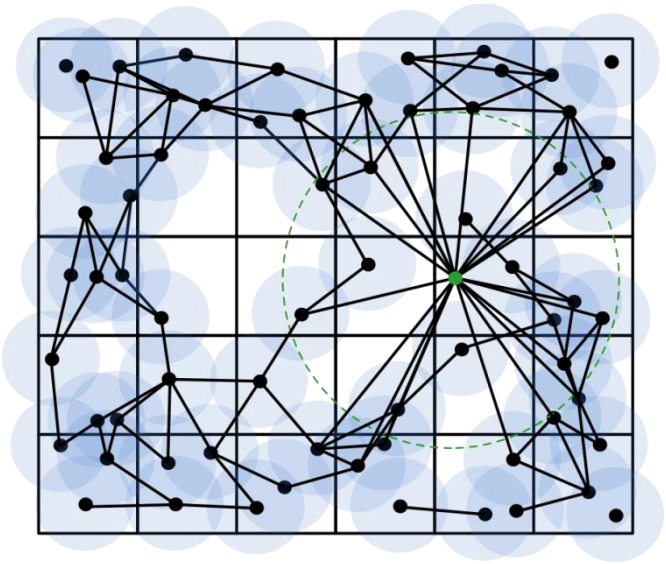

Figure 1.2: Randomized2D kinetoplast model with a supersaturated boundary (three nodes per edge gridbox) and one maxicircle (in green). This example assumes a link probability of 1 for any nodes with intersecting probability fields. Note that no nodes in the same gridbox link to each other regardless of probability field overlap. Note also that the maxi circle connects to all nodes whose probability fields intersect *the edge* of its field (the green dashed circle). Nodes with fields entirely within the maxicircle's field do not connect to the maxicircle.

Two additional features may be added to the base model: maxicircles and a supersaturated boundary. Maxicircles are nodes randomly placed in the gridspace with probability field radii larger – usually much larger – than the radii of other nodes. Currently, the code is written to guarantee a link between a maxicircle and any node whose probability field intersects the edge of the maxicircle's field, bypassing the probability integrator altogether. But, it is entirely possible to have a probability-function-based link probability for maxicircles with only some minor edits to the code. This was how it was done in the first version of the program.

A supersaturated boundary adds additional nodes beyond the usual one in all "boundary" gridboxes, the squares in the grid on the periphery of the gridspace. These nodes function exactly like any other nodes,

except that they are prevented from forming edges with any nodes in the same gridbox. See Fig 1.2 for an example of a randomized2D model with both a supersaturated boundary and a maxicircle.

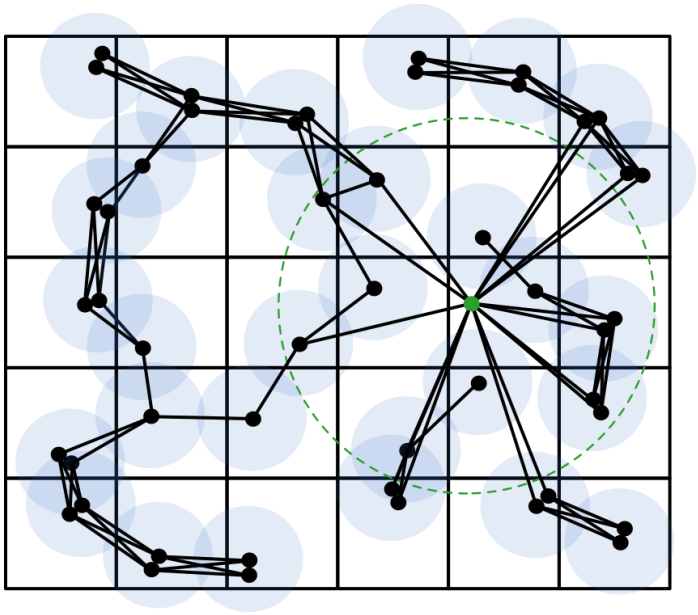

Figure 1.3: Randomized grid with a regular supersaturated boundary having two nodes per boundary stack and one maxi circle (in green). The nodes in each boundary stack are slightly offset to illustrate the structure, but in reality they would be directly atop one another. Note that nodes in the same stack do not link. Because this example assumes a guaranteed connection whenever probability fields intersect, both nodes in each stack have the same links, but this is not necessary.

As well as the randomly placed supersaturated boundary, there are two additional methods by which the boundary of a randomized grid can be supersaturated. The first is a “regular” boundary. This is a boundary in which, rather than having additional nodes placed at random in boundary boxes, new nodes are added with the same coordinates as the original node, forming “stacks” of nodes directly atop one another. In a regular boundary, each new node is treated individually, e.g. each has its probability function integrated over the intersect region with adjacent probability fields, and the existence of a link is determined using this probability. It is therefore possible to have some nodes in a stack link to a node in a different box when other nodes in the stack do not. Nodes in the same box/stack are still prevented from linking to each other. See Fig 1.3 for an example of a randomized grid with a regular supersaturated boundary and a maxi circle.

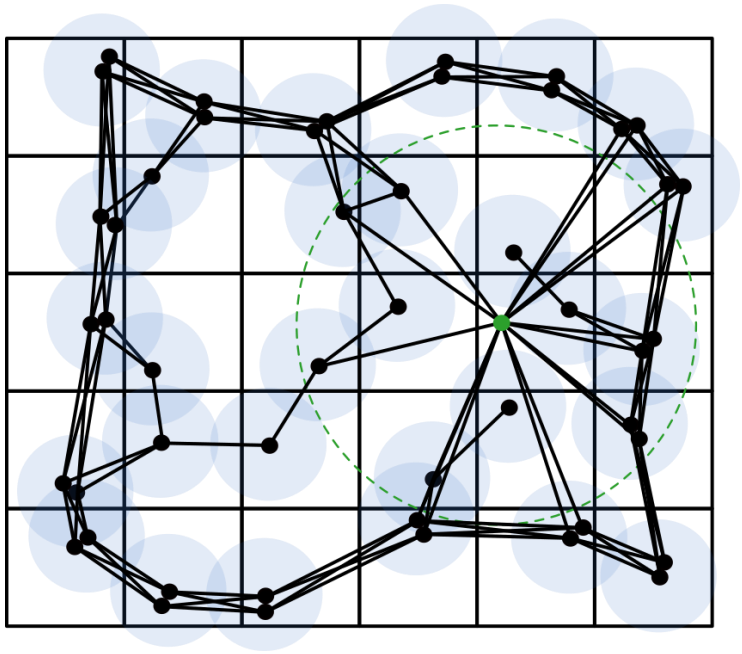

Figure 1.4: Randomized grid with a ringed supersaturated boundary having two nodes per boundary stack and one maxi circle (in green). The nodes in each boundary stack are slightly offset to illustrate the structure; in reality they would be directly atop one another. Note that this is identical to the regular boundary except that nodes in adjacent boundary boxes, even if their probability fields do not intersect, share a link. Additionally, in this type of model, it is a requirement that new nodes added to each stack link to nodes in different boxes if and only if the first, base node does.

The third type of supersaturated boundary is a “ringed” boundary. Like the regular boundary, new nodes are placed directly atop the old in stacks, but new nodes are considered clones of the original node in the boundary box. When testing for links from a boundary stack to other nodes, only this first node has its probability functions integrated. If it forms a link with a node, all other nodes in its stack follow suit. If it does not form a link, none of the other nodes in its stack will. Additionally, all nodes in a boundary stack are forced to link to all nodes in the two adjacent boundary stacks. For example, between two adjacent boundary stacks containing three nodes each, there will be nine edges, as every node in one stack links to every node in the other stack, forming a (local) complete bipartite graph. See Fig 1.4 for an example of a randomized grid with a ringed supersaturated boundary and a maxi circle.

Hex2D

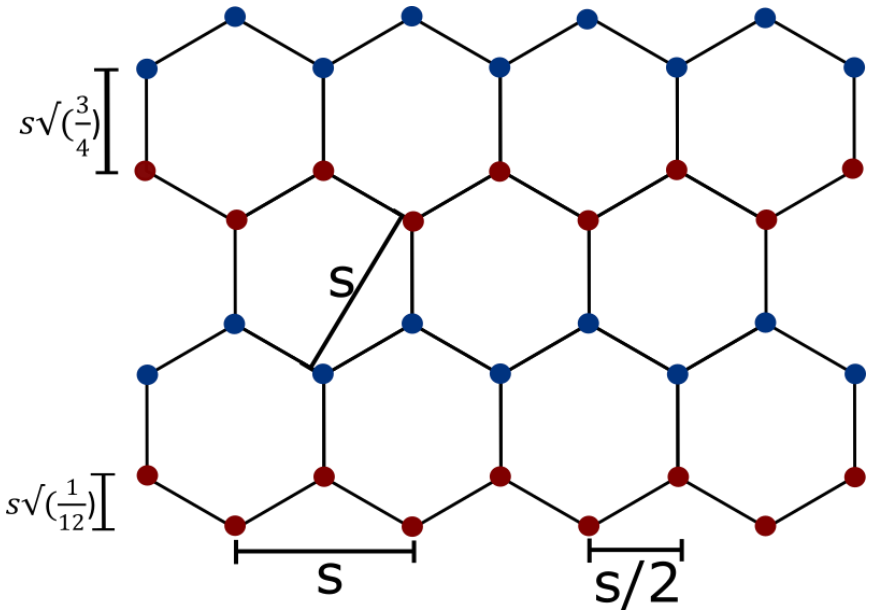

Figure 1.5: Basic Hex2D structure. The variable "s" is a user-selectable "separation" parameter, fundamentally the node spacing of the two underlying triangular lattices. Nodes are colored as they are added to the grid by the program; the two bottommost red rows are added, then the two blue rows, etc.

The two-dimensional hexagonal grid is a honeycomb structure composed of two offset triangular lattices, with the spacing between nodes depicted in Fig 1.5. The user enters two desired dimensions, "rows" and "columns", as well as the "separation" (defaults to 1) and the "radius" (defaults to 0.5). A "row" is defined to be a single line of points with identical x-coordinates, and a "column" a line of points with identical y-coordinates. So, in Fig 1.5, each colored band contains two rows. In order to ensure that the model always has the same qualitative shape as in Fig 1.5 (e.g. is composed of "complete" hexagons only, with no extra nodes dangling off the sides), the program will always ensure that there are an even number of rows and an odd number of columns. Nodes are generated in horizontal bands, the stripes of like color seen in Fig 1.5. There will be ceil([rows demanded by user]/2) such bands, where ceil() is the ceiling function, and ceil([columns demanded by user]/2)*2-1 columns.

The separation determines node placement as shown in Fig 1.5. The radius functions identically to the radius in a randomized grid, and determines the radius of the probability field surrounding each node. However, the links are not determined by integrating probability functions over intersect regions. Rather, non-maxicircle nodes are hard-coded to always link to their three nearest-neighbor stacks. This significantly reduces computation time and ensures a consistent structure. The radius is only used when determining connections to maxicircles.

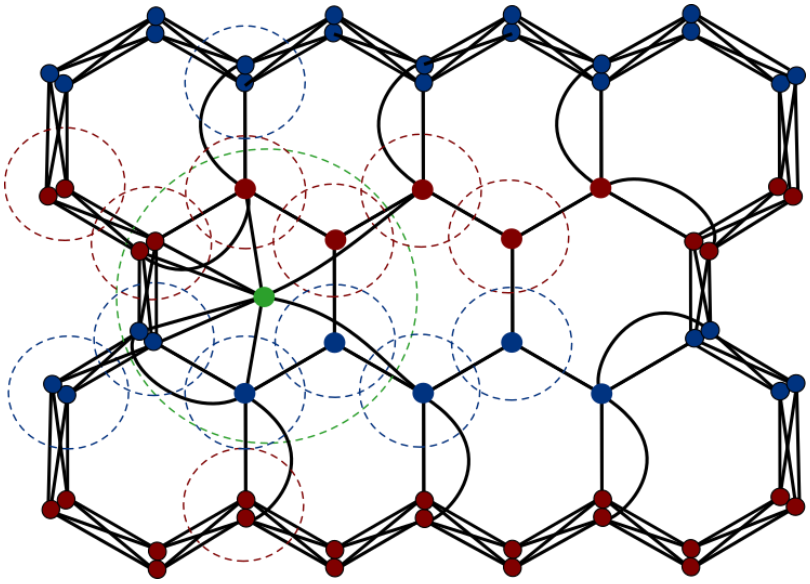

Figure 1.6: Hex2D kinetoplast model with a supersaturated boundary (two nodes per boundary stack) and one maxicircle (in green). Probability fields for nodes around the maxicircle are shown as dashed circles to illustrate how maxicircle links are determined. The two nodes in each boundary stack are slightly offset to illustrate the structure; in reality they are right on top of one another. Note that nodes within the same boundary stack do not link to each other.

Like the randomized model, the hex2D model can include maxicircles and/or a supersaturated boundary. Maxicircles function exactly as described in the discussion of the randomized2D model. The supersaturated boundary functions like a ringed boundary in a randomized grid. Any nodes that do not have exactly three neighbors or are adjacent to nodes with fewer than three neighbors are considered boundary nodes in the hex2D model. If the user wishes to supersaturate the boundary with *n* nodes per boundary stack, then for every boundary node (*n*-1) additional nodes are generated with the same coordinates. They form a stack of *n* points at every boundary location. Nodes in the same stack are prevented from forming a link with each other, but will each form an edge with all neighbors. See Fig 1.6 for an example of a hex2D model with a supersaturated boundary and a maxicircle.

Randomized3D

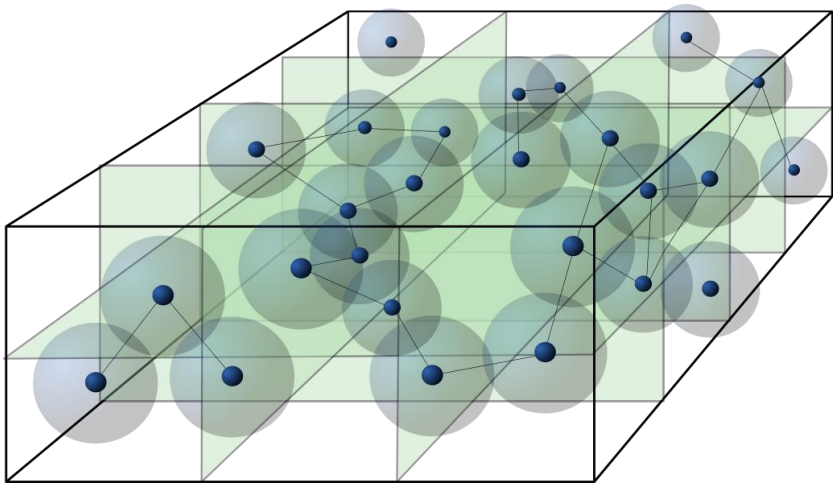

Figure 1.7: Randomized3D kinetoplast model with only the base grid. The light green rectangles with grey outlines are the dividing walls between different boxes. The dark blue spheres are nodes, surrounded by a transparent blue probability spheres. Each box has one node. Despite how it looks, the spheres are not getting smaller as you go back. They are just getting farther away. I would like to take this opportunity to point out I am not a graphic artist and ask that you do not look too closely at this janky attempt at 3D perspective.

The three-dimensional randomized model is a three-dimensional generalization of the randomized2D layout. The gridspace is divided in 1x1x1 boxes, and node(s) are randomly placed within each box by randomizing their x, y, and z coordinates. In place of the two-dimensional, circular probability fields of the randomized2D model, each node is surrounded by a three-dimensional probability sphere of user-defined radius. Link probability is determined by integrating two spheres' probability functions over the volume of intersection. As with the 2D model, the user may define the probability function in cylindrical coordinates manually in prob_funcs3D.py, although there are significant limitations to imposing an angular or z-dependence (see the Limitations section). Currently, probability is determined by integrating one sphere's probability function over the intersect region and dividing by the volume of the same sphere. The actual existence of a link is again determined by generating a random number from 0 to 1 and testing whether it is less than or equal to the link probability. See Fig 1.7 for an example of a randomized 3D model with two layers, four rows, and three columns.

A supersaturated boundary and maxispheres (3D maxicircles) can be added to a randomized3D model. Maxispheres work exactly like maxicircles, linking to any nodes with probability spheres intersecting the boundary of the maxisphere's field. The exception to this is other maxispheres; link probability between two maxispheres is determined by integrating the maxispheres' probability function over the region of intersection. The supersaturated boundary likewise functions identically to its 2D counterpart, with boundary gridboxes defined as those with maximal/minimal x-coordinates and maximal/minimal y-coordinates (so, the boundary of each plane of gridboxes, but not the entire top/bottom planes). The randomized3D model allows for only randomly placed supersaturated boundary nodes, not a regular or

ringed boundary. I wasn't brave enough to include either of these special features in Fig 1.7. You can probably imagine what they look like much more clearly than I can draw it.

Hex3D

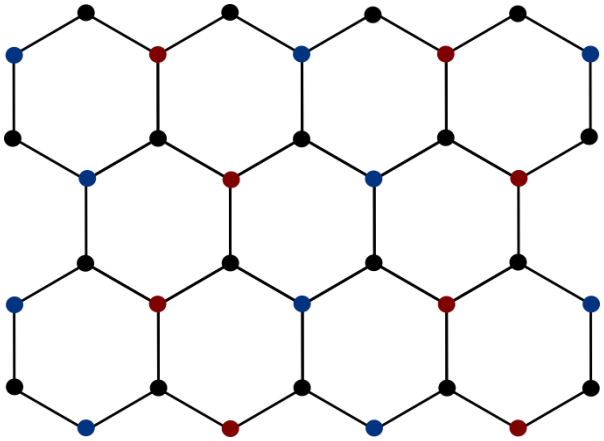

Figure 1.8: One plane of a hex3D kinetoplast model. Red nodes have links to the plane above (if it exists). Blue nodes have links to the plane below (if it exists). Black nodes have no link to another plane, as they do not lie directly above or below any nodes in the adjacent planes.

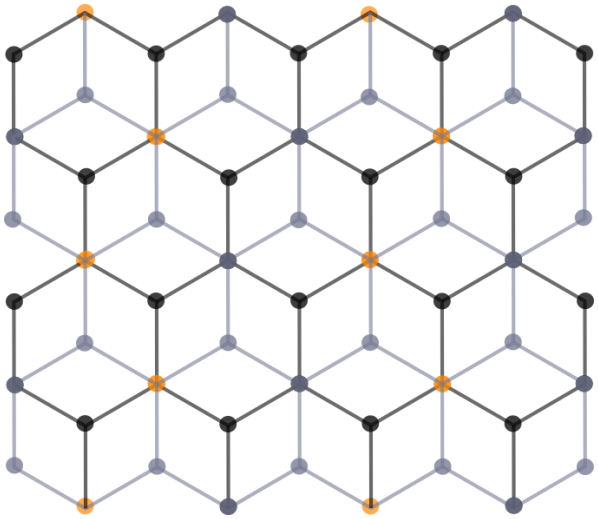

Figure 1.9: Two plane s of a hex3D kinetoplast model, seen from the top-down. The lower plane is grey, and the upper plane black. Nodes at which the two planes link to each other are in orange. Note that in each row that has connections, only every other node has a link between the two planes.

The three-dimensional hexagonal model is composed of layers of offset hex2D structure (called "planes" or "layers" depending upon which term I feel like using at the time) arranged like graphite. Each layer is connected to the one below it at every other aligned node, as shown in Fig 1.8. A top-down picture of two layers can be seen in Fig 1.9. The separation between each layer is one. Like hex2D, node links are not determined by link probability from integration of the probability function. Rather, connections are guaranteed to form as shown in Fig 1.8 and 1.9. Probability spheres with the probability function in

prob_funcs3D.py are still associated with each node, but are only used to determine links between maxispheres.

A supersaturated boundary and maxispheres may be added to a hex3D model. Maxispheres work identically to randomized3D maxispheres, automatically connecting to any nodes with probability spheres intersecting the maxisphere's boundary and having a probability of connecting to other maxispheres based on integration of the probability function. The supersaturated boundary is generated like the hex2D boundary, with any additional nodes having the same (x,y,z) coordinates as the other nodes in the boundary stack. The "boundary" is considered to be the boundary of each plane only (e.g. the entire top and entire bottom planes are **not** considered boundaries, but their edges are). As with the hex2D model, nodes in the same boundary stack do not link to each other, but if one node in a boundary stack links to another node, all nodes in the boundary stack link to that node.

Rectangular

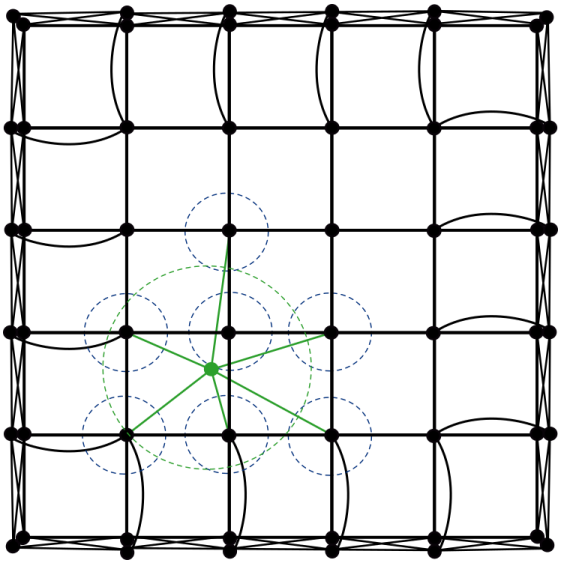

Figure 1.10: Rectangular kinetoplast model with a supersaturated boundary (two nodes per boundary stack) and one maxicircle (in green). Probability fields for nodes around the maxicircle are shown as dashed circles to illustrate how maxicircle links are determined. The two nodes in each boundary stack are slightly offset to illustrate the structure; in reality, they are right on top of one another. Note that nodes within the same boundary stack do not link to each other.

The rectangular model is a simple Z^2 integer lattice, where each node is a two-tuple of integer coordinates ((0,0); (1,0); (0,1); (1,1); (1,2), etc). Like the hex2D grid, links are guaranteed to form between a node and its four nearest-neighbor stacks. Each node still has a radius (defaults to 1) used in determining connections to maxicircles, and the separation parameter (also defaults to 1) determines the space between a node and its nearest neighbors.

Maxicircles and a supersaturated boundary can be added. Maxicircles work the same as in the hex2D and randomized2D models, and the supersaturated boundary works identically to the hex2D model.

Nodes added to the boundary have the same coordinates as the base nodes, and are grouped in "stacks." See Fig 1.10 for an example of a rectangular model with both a supersaturated boundary and a maxicircle.

Triangular

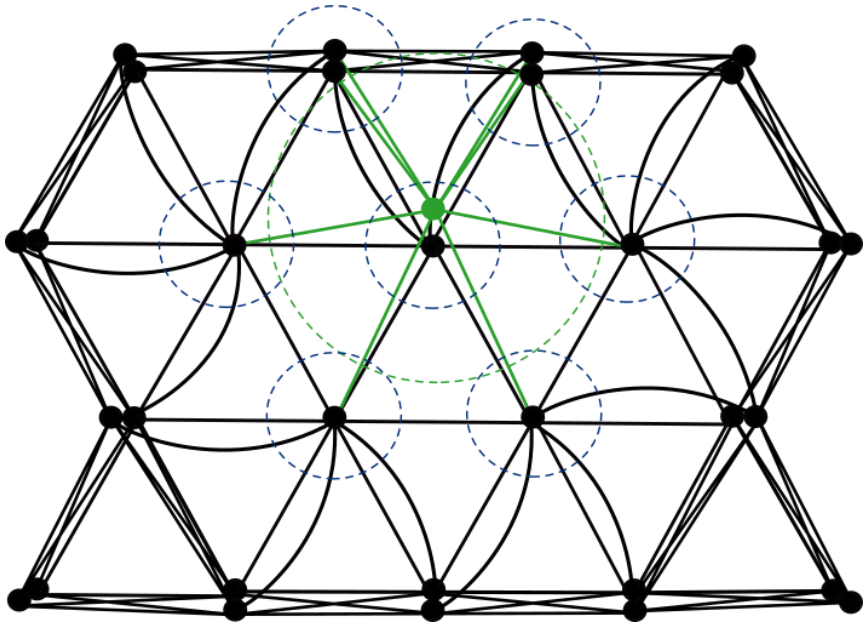

Figure 1.11: Triangular kinetoplast model with a supersaturated boundary (two nodes per boundary stack) and one maxicircle (in green). Probability fields for nodes around the maxicircle are shown as dashed circles to illustrate how maxicircle links are determined. The two nodes in each boundary stack are slightly offset to illustrate the structure; in reality, they are right on top of one another. Note that nodes within the same boundary stack do not link to each other.

The triangular model is a simple triangular lattice, where each row of nodes is identical, but shifted by ½ the separation in comparison to its neighboring rows. Each node thus has six nearest neighbors, two in the same row, two in the next row up, and two in the next row down. Like the hex2D grid, links are guaranteed to form between a node and its six nearest-neighbor stacks. Each node still has a radius (defaults to 1) used in determining connections to maxicircles, and the separation parameter (also defaults to 1) determines the space between a node and its nearest neighbors.

Maxicircles and a supersaturated boundary can be added. Maxicircles work the same as in the hex2D and randomized2D models, and the supersaturated boundary works identically to the hex2D model. Nodes added to the boundary have the same coordinates as the base nodes, and are grouped in "stacks." See Fig 1.11 for an example of a triangular model with both a supersaturated boundary and a maxicircle.

Dissolution Search

Once a kinetoplast network model has been generated, it can be dissolved. Dissolution is simulated by randomly removing nodes, and all edges incident on them, from the network model. The user may select the number of iterations, resolution, and threshold when starting a dissolution search. The number of iterations determines how many times a new model will be generated and a full dissolution search carried out before the program terminates. The resolution determines how many nodes are removed during a single "dissolution step." After each dissolution step, the size of every component in the network, down to individual isolated nodes, will be recorded and printed to an Excel file. This will proceed until every component has fewer nodes than the threshold value.

For example, if the user performed a 2-iteration, resolution-1, threshold-10 dissolution search, the program would generate a kinetoplast network model, record the sizes of all initial components, then remove nodes one at a time, recording component sizes after every removal. This would continue until there were no components with 10 or more nodes. Then, the output file would be closed, a new output file generated, a new kinetoplast model created, and the program would do exactly the same thing again for the second iteration.

Data Analysis Methods

There are currently six ways to view and interpret the results of a dissolution search: the raw output datafiles, component plots, dissolution plots, averaged dissolution plots, P' plots, and averaged P' plots. All but the component plots are generated using the raw output datafiles. In addition to the plots described here, the collection_plot.py script can be used to create a 3D plot showing individual/averaged dissolution or P' plots from multiple trials/series in one figure. This is not discussed here as it is simply a collection of multiple dissolution or P' plots.

Component Plots

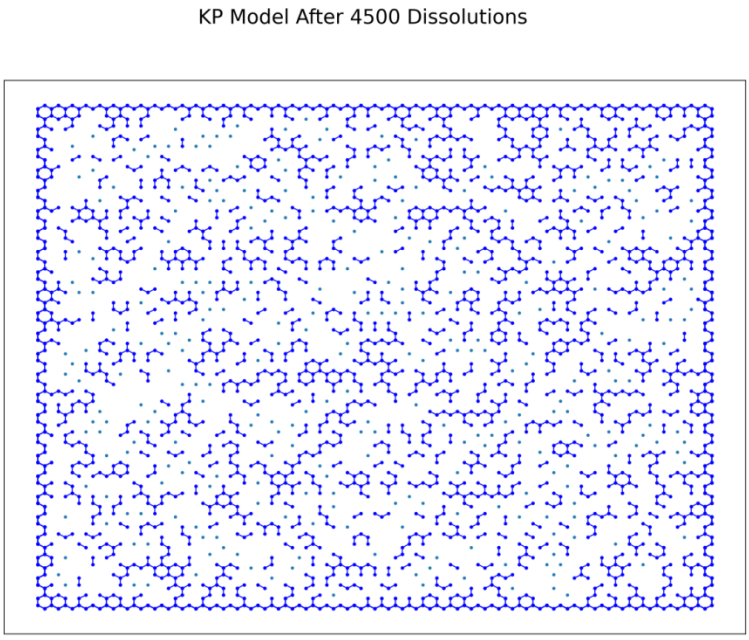

Figure 3.1: Component plot taken after 4500 dissolutions of a 100x100 hex2D model with no maxicircles and a supersaturated boundary with 10 nodes per boundary stack. Isolated nodes are a lighter shade of blue.

Component plots are snapshots of the gridspace during dissolution. They depict nodes as dots placed according to their coordinates with the origin in the bottom-left, and edges as lines linking the nodes.

These plots display the spatial distribution of components, which can be very useful in interpreting dissolution and P' plots. See Fig 3.1 for an example of a component plot.

Raw Output Datafiles

The Excel files created during a dissolution search contain one row for every dissolution step. Column A lists the number of individual dissolutions performed prior to collection of the row's data. The remaining columns each contain two-tuples, where the first element is component size (number of nodes) and the

second element is the number components which have that size. The tuples are listed in descending order of component size, so Column B lists the largest component, Column C the second largest, etc. One file is created per iteration of a dissolution search.

The raw data can be useful when the precise size of components is needed, and is used to generate the dissolution plots and P' plots.

Dissolution Plots

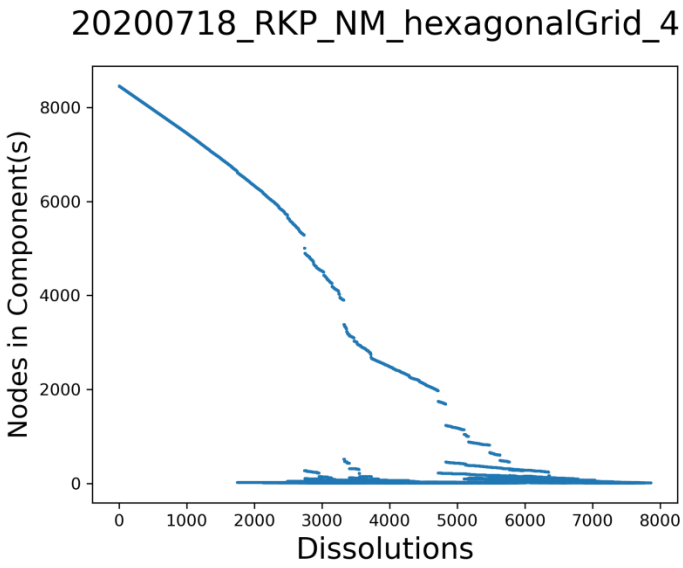

Figure 3.2: Dissolution plot for a 100x100 hex2D model with no maxicircles and a supersaturated boundary with 10 nodes per boundary stack. Only components with 10 or more nodes are represented. This is from the same simulation as Figure 3.1.

Dissolution plots track the evolution of components as they dissolve. The x-axis is the number of dissolutions, and the y-axis the number of nodes in the components present during a particular dissolution step. For every x-value, one point is generated for every component with size greater than the threshold. For example, if after 5000 dissolutions of a threshold-10 search there were 1 component of size 400, two of size 350, nine of size 12, two of size 9, and 200 of size 1, then the points (5000,400), (5000,350), and (5000, 12) would be added to the plot. The components of sizes 9 and 1 would be ignored as they are below the threshold.

As an additional optional feature, the point at which the boundary of a hexagonal, triangular, rectangular, or ringed randomized model develops its first hole (e.g. when the first boundary stack disappears entirely) can be marked in red on this plot, and the point at which the boundary breaks into two distinct, disconnected components (when the second boundary stack disappears entirely. I know technically the second stack to be obliterated could be right next to the first, but what are the chances of that?) can be marked in green. This is only possible if the flag_bound parameter is set to True throughout the model generation/dissolution process.

Dissolution plots are very useful for tracking the evolution of components as nodes are removed. Large jumps in component size can indicate the presence of a phase transition, when the order of the system dips suddenly as a critical node is lost and a large component breaks into smaller ones. The slope of the lines can indicate how well-connected the components are, as steep declines in component size indicate that the component is bleeding off large chucks of nodes with every dissolution. See Fig 3.2 for an example of a dissolution plot.

Averaged Dissolution Plots

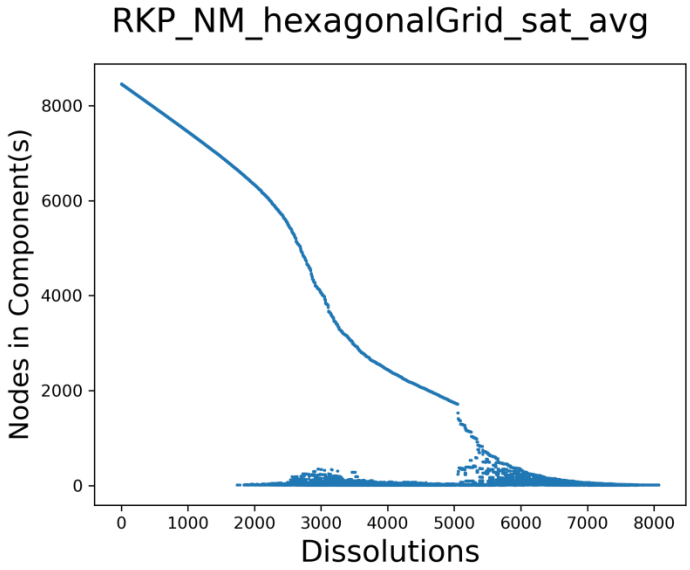

Figure 3.3: Averaged dissolution plot for 100x100 hex2D models with no maxicircles and a supersaturated boundary with 10 nodes per boundary stack. Only components with 10 or more nodes are represented. This plot is built using the results of 50 individual simulations using the same parameters (including the simulation used to generate Fig 3.2 and Fig 3.1).

Averaged dissolution plots display the same information as dissolution plots, but incorporate data from multiple simulations. It should be noted that they are not actually averages, but medians. I will continue to refer to it as an average, however, since I'm pretty sure "medianized" isn't actually a word. For each dissolution step (x-value), the list of component sizes is retrieved from the datafiles of all relevant simulations. The size of the largest component from each of these lists is then extracted. The median of the list of largest components is found, and the list of component sizes that the median came from is used to generate the points at the current x-value.

For example, say three simulations' datafiles were being aggregated, all using the same parameters with a threshold of 10. Suppose that at x=4000 dissolutions Simulation 1 had components of size 450, 200, 55, 10, and 1, Simulation 2 had components of size 350, 300, 250, 12, and 1, and Simulation 3 had sizes 250, 200, and 1. Then Simulation 2 would be used to generate points at this x-value, as the median of the list of largest components (250,350,450) is 350. So, the points (4000,350), (4000,300), (4000,250), and (4000,12) would be added to the plot (1 is below the threshold of 10, so (4000,1) is not added).

Suppose now that at x=4001, Simulation 1 has sizes 275, 200, 175, 55, 10, and 1, Simulation 2 has sizes 350, 250, 150, 75, 12, and 1, and Simulation 3 has sizes 250, 60, 40, 30, and 1. Then Simulation 1 would be used to generate points at this x-value, as the median of the list of largest components (250,275,350) is 275. So, the points (4000,275), (4000,200), (4000,175), (4000,55), and (4000,10) would be added to the plot.

The averaged dissolution plots trade depth for scope when compared to normal dissolution plots. The results of a large number of simulations can be viewed at once and outliers are suppressed, but at the cost of less-clearly-defined features and non-physical artefacts (like the random sprinkle of points at small y-values, as opposed to the well-defined component trails in individual dissolution plots). See Fig 3.3 for an example of an averaged dissolution plot.

P’ Plots

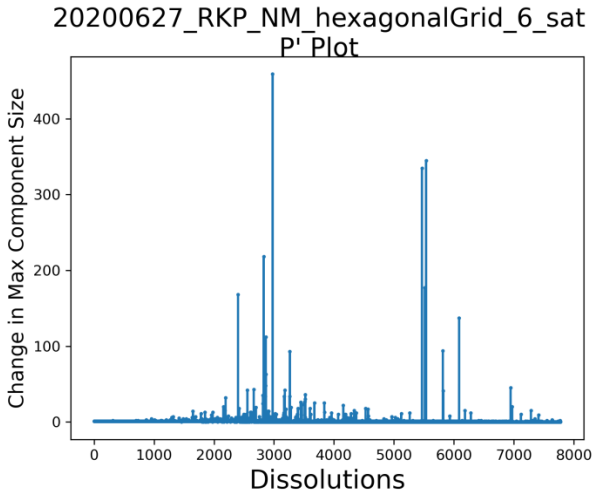

Figure 3.4: P’ plot for a 100x100 hex2D model with no maxicircles and a supersaturated boundary with 10 nodes per boundary stack. This is from a different simulation than Fig 3.1 and Fig 3.2, though it uses the same parameters.

P’ plots track the change in maximum component size as a function of the number of dissolutions. For each dissolution step (x-value), the size of the largest component is extracted. Call the number of dissolutions *n*, and the size of the largest component after *n* dissolutions *P(n)*, $n \in \{0,1,\dots,N\}$ where *N* is total number of dissolutions completed before the search terminated. Then the change in max component size is defined as $P'(n) = P(n) - P(n-1), n \in \{1,\dots,N\}$. This is the function that is plotted. Note that P’ is never negative, but may be zero, since the maximum component size always either stays the same or decreases with each dissolution.

For example, suppose that after 100 dissolutions the largest component has size 300, after 101 dissolutions the largest component has size 240, after 102 dissolutions the largest component has size 240, and after 103 dissolutions the largest component has size 175. Then the points (101,60), (102,0), and (103,65) would be added to the plot.

P’ plots are useful for pinpointing phase transitions, when large components finally disintegrate into many smaller components. It is also easy to see how these transitions happen; whether it is a single node disappearing that breaks down the largest component (a very tall, isolated spike), or whether it is several dissolutions in rapid succession that finally break it entirely apart (a series of tall spikes). See Fig 3.4 for an example of a P’ plot.

Averaged P’ Plots

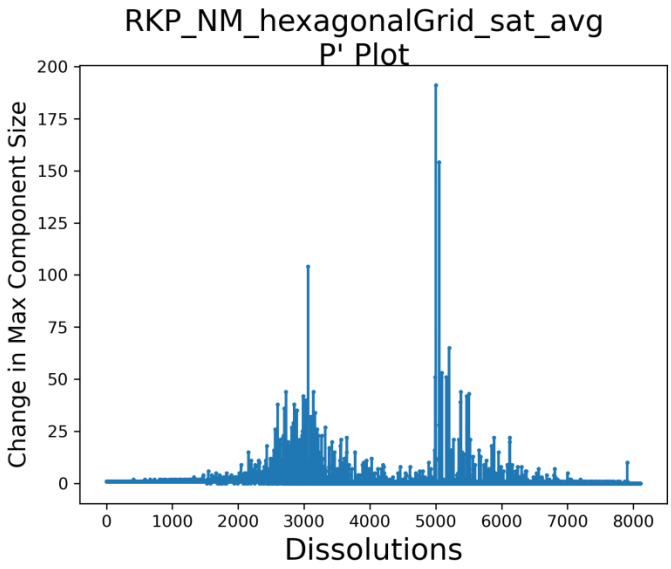

Figure 3.5: Averaged P’ plot for 100x100 hex2D models with no maxicircles and a supersaturated boundary with 10 nodes per boundary stack. This plot is built using the results of 30 individual simulations using the same parameters (including the simulation used to generate Fig 3.4).

Averaged P’ plots display the same information as P’ plots, but incorporate data from multiple simulations. As with averaged dissolution plots, it should be noted that they are not actually averages, but medians. For each dissolution step (x-value) *n*, the size of the largest component across all relevant simulations is extracted, and the median of this list is found. This median serves as $P(n)$. Otherwise, the definition of $P'(n)$ remains unchanged.

As with averaged dissolution plots, averaged P’ plots provide an easy way to view P’ results from many simulations at once, at the cost of reducing detail and introducing nonphysical behavior. As can be seen from comparing Fig 3.5 and Fig 3.4, the qualitative shape is similar. However, Fig 3.5 exhibits broad peaks in place of the sharp spikes seen in Fig 3.4. In reality, all P’ plots for simulations using these parameters resemble Fig 3.4’s spikes. The broadened peaks in Fig 3.5 are due to different simulations having spikes at different dissolution steps. This raises the median over a range of x-values in the averaged plot. This can be useful as it illustrates the range of dissolution steps at which a phase transition can occur but is not physical for any individual simulation.

General Usage

Usage depends upon what you want to do. There is no single, master program through which everything else is routed. Instead, what follows is a description of four included classes and stand-alone scripts that are used to run dissolution searches and generate plots. As previously mentioned, the tkinetoplast and rkinetoplast classes are not discussed due to their similarity to hkinetoplast, and the collection_plot.py script is not discussed as it simply applies the dissolution_plot_avg.py and delta_p_max_plot_avg.py algorithms to matplotlib's mplot3d package.

Class: kinetoplast

Creates a 2D randomized kinetoplast model and manages a dissolution search if desired.

Methods:

1. __init__(xdim,ydim,rad): The class is instantiated with three user-defined attributes and three autogenerated attributes.
    a. Input:
        i. xdim: Integer, the number of columns.
        ii. ydim: Integer, the number of rows.
        iii. rad: Float, the radius of all non-maxicircle probability fields.
    b. Attributes created/modified:
        i. xdim: Integer, the number of columns.
        ii. ydim: Integer, the number of rows.
        iii. rad: Float, the radius of all non-maxicircle probability fields.
        iv. grid: List of lists of lists of three-tuples of floats. Contains the location and radius of every point in the base randomized grid. Each three-tuple contains the x-coordinate, y-coordinate, and probability field radius of a single node, in that order. These tuples are grouped in lists of all nodes in the same gridbox. The gridbox lists are in turn grouped in a list representing one row of the grid, with the first element in the first column, the second in the second column, etc. The row lists are, in turn, grouped into a larger list of every row, starting with the bottom row. Generated using grid_gen.craft_grid().
        v. kpg: List. Empty when initialized.
        vi. kpn: String. Empty when initialized.
2. add_boundary(saturation,reg=False): Supersaturates the boundary of the kinetoplast object with additional nodes **beyond those already in the grid**. So, if each boundary gridbox already contains one node, calling kinetoplast.add_boundary(4) will result in five nodes per boundary gridbox. May be called multiple times.
    a. Input:
        i. saturation: Integer, the number of additional nodes to add to each boundary gridbox.

ii. reg: Boolean, defaults to False. If True, the new nodes will be added as a regular boundary, directly atop existing nodes.

b. Attributes created/modified:

i. grid: List of lists of lists of three-tuples of floats. Additional (x-coordinate, y-coordinate, radius) tuples are added to the gridbox list of each boundary gridbox. Modified using grid_gen.add_boundary() or, if reg=True, using grid_gen.add_reg_boundary().

3. add_maxi_circles(maxi_rad,amount): Adds maxicircles to the kinetoplast object. If called multiple times, the maxi_rad may be changed each time it is called.

a. Input:

i. maxi_rad: Float, the radius of the maxicircles' probability fields.

ii. amount: Integer, the number of maxicircles to add.

b. Attributes created/modified:

i. grid: List of lists of lists of three-tuples of floats. Additional (x-coordinate, y-coordinate, radius) tuples are added for each maxicircle. The coordinates are selected randomly within the gridspace, and the new nodes are placed in the proper gridbox lists. Modified using grid_gen.add_maxi_circles().

4. compile_graph(ringed=False): Turns the current grid attribute into an adjacency matrix and networkx network object using the probability function in prob_funcs.py to determine link probability. May be called multiple times; each time, kpg and kpn will be regenerated using the **current** grid attribute and the probability function from prob_funcs.py **that existed when the kinetoplast module was first imported**. When changing the probability function, the shell must also be restarted.

a. Input:

i. ringed: Boolean, defaults to false. If True, creates a ringed boundary. Unpredictable things may happen if ringed is called on a non-regular boundary, including the program crashing.

b. Attributes created/modified:

i. kpg: List of lists of 1s and 0s. This is the adjacency matrix for the network model of the kinetoplast. Nodes are listed in the order they appear in grid; the first row of kpg gives links from the node in the [0][0][0] index of the grid (bottom row, first column, first node in the gridbox list) to all other nodes, the second row of kpg gives links from the node in the [0][0][1] index of the grid (bottom row, first column, second node in the gridbox list) to all other nodes, etc. Generated using grid_gen.to_graph().

ii. kpn: networkx network object. This is the network representation of the kinetoplast, and is used in dissolution searches. Each node retains its spatial coordinate and radius tuple (x,y,rad) as a property called 'Coords'. Generated using graph_funcs.to_network().

5. compile_graph_alg(): Debug method that does the same thing as compile_graph(), but uses algebraic methods to determine link probability rather than integration. Only works for very specific probability functions and parameters. Don't worry about it.

6. plot_grid(): Creates a pyplot of the current grid attribute and displays it as a pop-up. Shows only nodes, not the underlying grid or links. Image is not saved unless the user does so manually. Uses grid_gen.grid_visualizer().
    a. Inputs: None.
    b. Attributes created/modified: None.
7. plot_graph(): Creates a pyplot of the current kpn attribute and displays it as a pop-up. Shows nodes and links. Image is not saved unless the user does so manually. Uses graph_funcs.graph_visualizer().
    a. Inputs: None.
    b. Attributes created/modified: None.
8. min_deg(): Returns the minimum degree of the current kpn attribute. Uses graph_funcs.min_deg().
    a. Inputs: None.
    b. Attributes created/modified: None.
9. avg_deg(): Returns the average degree of the current kpn attribute. Uses graph_funcs.avg_deg().
    a. Inputs: None.
    b. Attributes created/modified: None.
10. dissolve(resolution, thresh, excel_name): Performs a dissolution search on the current kpn attribute and outputs both a raw output datafile and a dissolution plot. This permanently dissolves the kpn object; it has to be regenerated using compile_graph() if another search is to be performed.
    a. Input:
        i. resolution: Integer, the resolution of the search, e.g. how many dissolutions are performed per dissolution step.
        ii. thresh: Integer, the component size threshold of the search. The search will terminate when the size of all remaining components is less than this value. Additionally, all components of size less than the threshold will be omitted from the autogenerated dissolution plot.
        iii. excel_name: String, the name of the output Excel file. **This does not include the directory or .xlsx extension**. You must manually change the desired directory of the Excel output file in **both** bf_search.py and dissolution_plot.py, and the desired directory of the dissolution plot image in dissolution_plot.py. The name of the plot image will be excel_name + "_plot.png".
    b. Attributes created/modified:
        i. kpn: networkx network object. kpn is left in its dissolved state, with only those nodes still present when the dissolution search terminated. It is possible to create an exact copy of the kpn network object, without re-randomizing link creation, after a dissolution search since grid and kpg are not altered. However, this can only be done by manually feeding the grid attribute into the function grid_gen.to_graph() and collecting the object "idm" outputted by this function, then manually feeding the idm object and the grid and kpg attributes into

graph_funcs.to_network(). This will **not** regenerate the kpn attribute, but will create an independent clone of the original kpn network object.

c. File output:
    i. Raw output datafile: Excel spreadsheet with layout described in the Data Analysis Methods section. The file will be deposited in the directory entered in bf_search.py, and be named excel_name+".xlsx".
    ii. Dissolution plot: Dissolution plot with layout described in the Data Analysis Methods section. The file will be deposited in the directory entered in dissolution_plot.py, and be named excel_name + "_plot.png".

Class: hkinetoplast

Creates a hex2D kinetoplast model and manages a dissolution search if desired.

Methods:

1. __init__(xdim,ydim,flag_bound=False,rad=0.5,sep=1): The class is instantiated with five user-defined attributes and six autogenerated attributes.
    a. Input:
        i. xdim: Integer, the number of columns.
        ii. ydim: Integer, the number of rows.
        iii. flag_bound: Boolean, defaults to False. If True, an extra entry will be added to each node tuple, a Boolean that is True if a node is in a boundary stack and False otherwise. This parameter must be set to True if dissolve_flag_bound() is to be used.
        iv. rad: Float, defaults to 0.5. The radius of all non-maxicircle probability fields. Only determines which nodes link to maxicircles.
        v. sep: Float, defaults to 1.The distance between nodes in each of the two underlying triangular lattices.
    b. Attributes created/modified:
        i. xdim: Integer, the number of columns.
        ii. ydim: Integer, the number of rows.
        iii. rad: Float, the radius of all non-maxicircle probability fields.
        iv. sep: Float, the distance between nodes in each of the two underlying triangular lattices.
        v. grid: List of lists of lists of three-tuples of floats. Contains the location and radius of every point in the base randomized grid. Each three-tuple contains the x-coordinate, y-coordinate, and probability field radius of a single point, in that order. These tuples are grouped in lists of all nodes in the same stack. The stack lists are in turn grouped in a list representing one row of the grid, with the first element in the first column, the second in the second column, etc. The row lists are, in turn, grouped into a larger list of every row, starting with the bottom row. Generated using grid_gen.hexagonal_grid().
        vi. kpg: List. Empty when initialized.
        vii. kpn: String. Empty when initialized.
        viii. maxi: Boolean. False when initialized.
        ix. supersat: Boolean. False when initialized.
        x. flag_bound: Boolean. If True, an extra entry will be added to each node tuple, a Boolean that is True if a node is in a boundary stack and False otherwise.
        xi. sat: Integer. The number of nodes in each boundary stack. Equal to one when initialized, since there is only one node in each boundary stack at the start.

2. add_boundary(saturation): Supersaturates the boundary of the kinetoplast object with additional nodes **beyond those already in the grid**. So, if each boundary stack already contains one node, calling hkinetoplast.add_boundary(4) will result in five nodes per boundary stack. May be called multiple times.
    a. Input:
        i. saturation: Integer, the number of additional nodes to add to each boundary stack.
    b. Attributes created/modified:
        i. grid: List of lists of lists of three-tuples of floats. Additional (x-coordinate, y-coordinate, radius) tuples are added to the stack list of each boundary stack. Modified using grid_gen.hexagonal_boundary().
        ii. supersat: Boolean. Set to True.
        iii. sat: Integer. The saturation input parameter is added to whatever value this attribute previously had.
3. add_maxi_circles(maxi_rad,amount): Adds maxicircles to the kinetoplast object. If called multiple times, the maxi_rad may be changed each time it is called.
    a. Input:
        i. maxi_rad: Float, the radius of the maxicircles' probability fields.
        ii. amount: Integer, the number of maxicircles to add.
    b. Attributes created/modified:
        i. grid: List of lists of lists of three-tuples of floats. Additional (x-coordinate, y-coordinate, radius) tuples are added for each maxicircle. The coordinates are selected randomly within the gridspace, and the new nodes are placed in the closest row lists. Modified using grid_gen.add_maxi_circles().
        ii. maxi: Boolean. Set to True.
4. compile_graph(): Turns the current grid attribute into an adjacency matrix and networkx network object. May be called multiple times; each time, kpg and kpn will be regenerated using the **current** grid attribute and the probability function from prob_funcs.py **that existed when the hkinetoplast module was first imported**. When changing the probability function, the shell must also be restarted.
    a. Input: None
    b. Attributes created/modified:
        i. kpg: List of lists of 1s and 0s. This is the adjacency matrix for the network model of the kinetoplast. Nodes are listed in the order they appear in grid; the first row of kpg gives links from the node in the [0][0][0] index of the grid (bottom row, first column, first node in the stack) to all other nodes, the second row of kpg gives links from the node in the [0][0][1] index of the grid (bottom row, first column, second node in the stack) to all other nodes, etc. Generated using grid_gen.to_graph_hex().
        iii. kpn: networkx network object. This is the network representation of the kinetoplast and is used in dissolution searches. Each node retains its spatial

coordinate and radius tuple (x,y,rad) as a property called 'Coords'. Generated using graph_funcs.to_network().

5. plot_grid(): Creates a pyplot of the current grid attribute and displays it as a pop-up. Shows only nodes, not the underlying grid or links. Image is not saved unless the user does so manually. Uses grid_gen.grid_visualizer().
    a. Inputs: None.
    b. Attributes created/modified: None.
6. plot_graph(): Creates a pyplot of the current kpn attribute and displays it as a pop-up. Shows nodes and links. Image is not saved unless the user does so manually. Uses graph_funcs.graph_visualizer().
    a. Inputs: None.
    b. Attributes created/modified: None.
7. min_deg(): Returns the minimum degree of the current kpn attribute. Uses graph_funcs.min_deg().
    a. Inputs: None.
    b. Attributes created/modified: None.
8. avg_deg(): Returns the average degree of the current kpn attribute. Uses graph_funcs.avg_deg().
    a. Inputs: None.
    b. Attributes created/modified: None.
9. dissolve(resolution, thresh, excel_name): Performs a dissolution search on the current kpn attribute and outputs both a raw output datafile and a dissolution plot. This permanently dissolves the kpn object; it has to be regenerated using compile_graph() if another search is to be performed.
    a. Input:
        i. resolution: Integer, the resolution of the search, e.g. how many dissolutions are performed per dissolution step.
        ii. thresh: Integer, the component size threshold of the search. The search will terminate when the size of all remaining components is less than this value. Additionally, all components of size less than the threshold will be omitted from the autogenerated dissolution plot.
        iii. excel_name: String, the name of the output Excel file. **This does not include the directory or .xlsx extension**. You must manually change the desired directory of the Excel output file in **both** bf_search.py and dissolution_plot.py, and the desired directory of the dissolution plot image in dissolution_plot.py. The name of the plot image will be excel_name + "_plot.png".
    b. Attributes created/modified:
        i. kpn: networkx network object. kpn is left in its dissolved state, with only those nodes still present when the dissolution search terminated. It is possible to create an exact copy of the kpn network object, without re-randomizing link creation, after a dissolution search since grid and kpg are not altered. See the kinetoplast.dissolve() entry for a description of how to do this.

c. File output:
    i. Raw output datafile: Excel spreadsheet with layout described in the Data Analysis Methods section. The file will be deposited in the directory entered in bf_search.py, and be named excel_name + ".xlsx".
    ii. Dissolution plot: Dissolution plot with layout described in the Data Analysis Methods section. The file will be deposited in the directory entered in dissolution_plot.py, and be named excel_name + "_plot.png".

10. dissolve_with_pics(resolution, thresh, excel_name): Performs a dissolution search on the current kpn attribute. Functions identically to dissolve(), but also exports component plots each time a particular number of dissolutions have occurred. The intervals at which component plots are created are pre-defined within the definition of dissolve_with_pics(), and depend upon the existence/nonexistence of a supersaturated boundary and/or maxicircles.
    a. Input:
        i. resolution: Integer, the resolution of the search, e.g. how many dissolutions are performed per dissolution step.
        ii. thresh: Integer, the component size threshold of the search. The search will terminate when the size of all remaining components is less than this value. Additionally, all components of size less than the threshold will be omitted from the autogenerated dissolution plot.
        iii. excel_name: String, the name of the output Excel file. **This does not include the directory or .xlsx extension**. You must manually change the desired directory of the Excel output file in bf_search.py and dissolution_plot.py, the desired directory of the dissolution plot image in dissolution_plot.py, and the desired directory of the component plots in the definition of this method. The name of the plot image will be excel_name + "_plot.png", and all component plots will be deposited in a newly-created folder called excel_name+"_componentPlots".
    b. Attributes created/modified:
        i. kpn: networkx network object. kpn is left in its dissolved state, with only those nodes still present when the dissolution search terminated. It is possible to create an exact copy of the kpn network object, without re-randomizing anything, after a dissolution search as grid and kpg are not altered. See the kinetoplast.dissolve() entry for a description of how to do this.
    c. File output:
        i. Raw output datafile: Excel spreadsheet with layout described in the Data Analysis Methods section. The file will be deposited in the directory entered in bf_search.py, and be named excel_name+".xlsx".
        ii. Dissolution plot: Dissolution plot with layout described in the Data Analysis Methods section. The file will be deposited in the directory entered in dissolution_plot.py, and be named excel_name + "_plot.png".
        iii. Component plots: Component plots with layout described in the Data Analysis Methods section. One plot will be generated for every dissolution step flagged in the definition of this method. They will be named "after"+[dissolution

step]+"dissolutions_componentPlot.png", and be deposited in a folder titled excel_name+"_componentPlots" at the directory listed in the definition of this method.

11. dissolve_flag_bound(self,resolution,thresh,excel_name): Performs a dissolution search on the current kpn attribute. Functions identically to dissolve(), but also records when the first and second hole appears in the model's boundary. This method will only work if the flag_bound attribute was set to True when the class was instantiated.
    a. Input:
        i. resolution: Integer, the resolution of the search, e.g. how many dissolutions are performed per dissolution step.
        ii. thresh: Integer, the component size threshold of the search. The search will terminate when the size of all remaining components is less than this value. Additionally, all components of size less than the threshold will be omitted from the autogenerated dissolution plot.
        iii. excel_name: String, the name of the output Excel file. **This does not include the directory or .xlsx extension**. You must manually change the desired directory of the Excel output file in **both** bf_search.py and dissolution_plot.py, and the desired directory of the dissolution plot image in dissolution_plot.py. The name of the plot image will be excel_name + "_plot.png".
    b. Attributes created/modified:
        i. kpn: networkx network object. kpn is left in its dissolved state, with only those nodes still present when the dissolution search terminated. It is possible to create an exact copy of the kpn network object, without re-randomizing anything, after a dissolution search as grid and kpg are not altered. See the kinetoplast.dissolve() entry for a description of how to do this.
    c. File output:
        i. Raw output datafile: Excel spreadsheet with layout described in the Data Analysis Methods section. Additionally, the dissolutions step at which the first hole develops in the boundary will be noted by printing the phrase "Boundary Breaks Here" after the list of component sizes, and the step at which the second hole appears is noted with the phrase "Boundary Splits in Two Here". The file will be deposited in the directory entered in bf_search.py, and be named excel_name+".xlsx".
        ii. Dissolution plot: Dissolution plot with layout described in the Data Analysis Methods section. The point at which the first hole in the boundary develops will be shown in red, and the point at which the second develops will be shown in green. The file will be deposited in the directory entered in dissolution_plot.py, and be named excel_name + "_plot.png".

Class: kinetoplast3D

Creates a 3D randomized kinetoplast model and manages a dissolution search if desired.

Methods:

1. __init__(xdim,ydim,rad): The class is instantiated with four user-defined attributes and three autogenerated attributes.
    a. Input:
        i. xdim: Integer, the number of columns.
        ii. ydim: Integer, the number of rows.
        iii. zdim: Integer, the number of planes.
        iv. rad: Float, the radius of all non-maxisphere probability fields
    b. Attributes created/modified:
        i. xdim: Integer, the number of columns.
        ii. ydim: Integer, the number of rows.
        iii. zdim: Integer, the number of planes.
        iv. rad: Float, the radius of all non-maxicircle probability fields.
        v. grid: List of lists of lists of lists of four-tuples of floats. Confused yet? Contains the location and radius of every point in the base randomized grid. Each four-tuple contains the x-coordinate, y-coordinate, z-coordinate, and probability field radius of a single point, in that order. These tuples are grouped in lists of all nodes in the same gridbox. The gridbox lists are in turn grouped in a list representing one row of the plane, where the first element is in the first column, the second in the second column, etc. The row lists are, in turn, grouped into a larger list of every row in one plane, starting with the bottom row. Finally, planes are organized in the top-level list, with the bottom plane first. Generated using grid_gen3D.craft_grid3D().
        vi. kpg: List. Empty when initialized.
        vii. kpn: String. Empty when initialized.
2. add_boundary(saturation): Supersaturates the boundary of the kinetoplast object with additional nodes **beyond those already in the grid**. So, if each boundary gridbox already contains one node, calling kinetoplast3D.add_boundary(4) will result in five nodes per boundary gridbox. May be called multiple times.
    a. Input:
        i. saturation: Integer, the number of additional nodes to add to each boundary gridbox.
    b. Attributes created/modified:
        i. grid: List of lists of lists of lists of four-tuples of floats. Additional (x-coordinate, y-coordinate, z-coordinate, radius) tuples are added to the gridbox list of each boundary gridbox. Modified using grid_gen3D.add_boundary().

3. add_maxi_circles(maxi_rad,amount): Adds maxispheres to the kinetoplast object. If called multiple times, the maxi_rad may be changed each time it is called.
    a. Input:
        i. maxi_rad: Float, the radius of the maxispheres' probability fields.
        ii. amount: Integer, the number of maxispheres to add.
    b. Attributes created/modified:
        i. grid: List of lists of lists of lists of four-tuples of floats. Additional (x-coordinate, y-coordinate, z-coordinate, radius) tuples are added for each maxisphere. The coordinates are selected randomly within the gridspace, and the new nodes are placed in the proper gridbox lists. Modified using grid_gen3D.add_maxi_circles().
4. compile_graph(): Turns the current grid attribute into an adjacency matrix and networkx network object using the probability function in prob_funcs3D.py to determine link probability. May be called multiple times; each time, kpg and kpn will be regenerated using the **current** grid attribute and the probability function from prob_funcs3D.py **that existed when the kinetoplast3D module was first imported**. When changing the probability function, the shell must also be restarted.
    a. Input: None
    b. Attributes created/modified:
        i. kpg: List of lists of 1s and 0s. This is the adjacency matrix for the network model of the kinetoplast. Nodes are listed in the order they appear in the grid; the first row of kpg gives links from the node in the [0][0][0][0] index of the grid (bottom plane, bottom row, first column, first node in the gridbox list) to all other nodes, the second row of kpg gives links from the node in the [0][0][0][1] index of the grid (bottom plane, bottom row, first column, second node in the gridbox list) to all other nodes, etc. Generated using grid_gen3D.to_graph().
        ii. kpn: networkx network object. This is the network representation of the kinetoplast, and is used in dissolution searches. Each node retains its spatial coordinate and radius tuple (x,y,z,rad) as a property called 'Coords'. Generated using graph_funcs3D.to_network().
5. min_deg(): Returns the minimum degree of the current kpn attribute. Uses graph_funcs3D.min_deg().
    a. Inputs: None.
    b. Attributes created/modified: None.
6. avg_deg(): Returns the average degree of the current kpn attribute. Uses graph_funcs3D.avg_deg().
    a. Inputs: None.
    b. Attributes created/modified: None.
7. dissolve(resolution, thresh, excel_name): Performs a dissolution search on the current kpn attribute and outputs both a raw output datafile and a dissolution plot. This permanently dissolves the kpn object; it has to be regenerated using compile_graph() if another search is to be performed.

a. Input:
    i. resolution: Integer, the resolution of the search, e.g. how many dissolutions are performed per dissolution step.
    ii. thresh: Integer, the component size threshold of the search. The search will terminate when the size of all remaining components is less than this value. Additionally, all components of size less than the threshold will be omitted from the autogenerated dissolution plot.
    iii. excel_name: String, the name of the output Excel file. **This does not include the directory or .xlsx extension**. You must manually change the desired directory of the Excel output file in **both** bf_search3D.py and dissolution_plot3D.py, and the desired directory of the dissolution plot image in dissolution_plot3D.py. The name of the plot image will be excel_name + "_plot.png".
b. Attributes created/modified:
    i. kpn: networkx network object. kpn is left in its dissolved state, with only those nodes still present when the dissolution search terminated. It is possible to create an exact copy of the kpn network object, without re-randomizing link creation, after a dissolution search as grid and kpg are not altered. However, this can only be done by manually feeding the grid attribute into the function grid_gen3D.to_graph() and collecting the object "idm" outputted by this function, then manually feeding the idm object and the grid and kpg attributes into graph_funcs3D.to_network(). This will **not** regenerate the kpn attribute, but will create an independent clone of the original kpn network object.
c. File output:
    i. Raw output datafile: Excel spreadsheet with layout described in the Data Analysis Methods section. The file will be deposited in the directory entered in bf_search3D.py, and be named excel_name+".xlsx".
    ii. Dissolution plot: Dissolution plot with layout described in the Data Analysis Methods section. The file will be deposited in the directory entered in dissolution_plot3D.py, and be named excel_name + "_plot.png".

Class: hkinetoplast3D

Creates a hex3D kinetoplast model and manages a dissolution search if desired.

Methods:

1. __init__(xdim,ydim,zdim): The class is instantiated with three user-defined attributes and three autogenerated attributes.
    a. Input:
        i. xdim: Integer, the number of columns.
        ii. ydim: Integer, the number of rows.
        iii. zdim: Integer, the number of planes.
    b. Attributes created/modified:
        i. xdim: Integer, the number of columns.
        ii. ydim: Integer, the number of rows.
        iii. zdim: Integer, the number of planes.
        iv. grid: List of lists of lists of lists of four-tuples of floats. Contains the location and radius of every point in the grid. Each three-tuple contains the x-coordinate, y-coordinate, z-coordinate, and probability field radius (hard coded to be 1) of a single node, in that order. These tuples are grouped in lists of all nodes in the same stack. The stack lists are in turn grouped in a list representing two rows of the grid. The row lists are, in turn, grouped into a larger list of every row in one plane, starting with the bottom row. Finally, planes are organized in the top-level list with the bottom plane first. Generated using grid_gen3D.graphite_grid().
        v. kpg: List. Empty when initialized.
        vi. kpn: String. Empty when initialized.
2. add_boundary(saturation): Supersaturates the boundary of the kinetoplast object with additional nodes **beyond those already in the grid**. So, if each boundary stack already contains one node, calling hkinetoplast3D.graphite_boundary(4) will result in five nodes per boundary stack. May be called multiple times.
    a. Input:
        i. saturation: Integer, the number of additional nodes to add to each boundary stack.
    b. Attributes created/modified:
        i. grid: List of lists of lists of lists of four-tuples of floats. Additional (x-coordinate, y-coordinate, z-coordinate, radius) tuples are added to the list of each boundary stack. Modified using grid_gen3D.graphite_boundary().
3. add_maxi_circles(maxi_rad,amount): Adds maxispheres to the kinetoplast object. If called multiple times, the maxi_rad may be changed each time it is called.
    a. Input:
        i. maxi_rad: Float, the radius of the maxispheres' probability fields.

ii. amount: Integer, the number of maxispheres to add.

b. Attributes created/modified:

i. grid: List of lists of lists of lists of four-tuples of floats. Additional (x-coordinate, y-coordinate, z-coordinate, radius) tuples are added for each maxisphere. The coordinates are selected randomly within the gridspace and each new tuple is placed in the closest stack. Modified using grid_gen3D.graphite_maxi_circles().

4. compile_graph(): Turns the current grid attribute into an adjacency matrix and networkx network object. May be called multiple times; each time, kpg and kpn will be regenerated using the **current** grid attribute.

a. Input: None

b. Attributes created/modified:

i. kpg: List of lists of 1s and 0s. This is the adjacency matrix for the network model of the kinetoplast. Nodes are listed in the order they appear in grid; the first row of kpg gives links from the node in the [0][0][0][0] index of the grid (bottom plane, bottom row, first column, first node in the stack) to all other nodes, the second row of kpg gives links from the node in the [0][0][0][1] index of the grid (bottom plane, bottom row, first column, second node in the stack) to all other nodes, etc. Generated using grid_gen3D.to_graph_graphite ().

ii. kpn: networkx network object. This is the network representation of the kinetoplast, and is used in dissolution searches. Each node retains its spatial coordinate and radius tuple (x,y,z,rad) as a property called 'Coords'. Generated using graph_funcs3D.to_network().

5. min_deg(): Returns the minimum degree of the current kpn attribute. Uses graph_funcs3D.min_deg().

a. Inputs: None.

b. Attributes created/modified: None.

6. avg_deg(): Returns the average degree of the current kpn attribute. Uses graph_funcs3D.avg_deg().

a. Inputs: None.

b. Attributes created/modified: None.

7. dissolve(resolution, thresh, excel_name): Performs a dissolution search on the current kpn attribute and outputs both a raw output datafile and a dissolution plot. This permanently dissolves the kpn object; it has to be regenerated using compile_graph() if another search is to be performed.

a. Input:

i. resolution: Integer, the resolution of the search, e.g. how many dissolutions are performed per dissolution step.

ii. thresh: Integer, the component size threshold of the search. The search will terminate when the size of all remaining components is less than this value. Additionally, all components of size less than the threshold will be omitted from the autogenerated dissolution plot.

iii. excel_name: String, the name of the output Excel file. **This does not include the directory or .xlsx extension**. You must manually change the desired directory of the Excel output file in **both** bf_search3D.py and dissolution_plot3D.py, and the desired directory of the dissolution plot image in dissolution_plot3D.py. The name of the plot image will be excel_name + "_plot.png".

b. Attributes created/modified:
   i. kpn: networkx network object. kpn is left in its dissolved state, with only those nodes still present when the dissolution search terminated. It is possible to create an exact copy of the kpn network object, without re-randomizing link creation, after a dissolution search as grid and kpg are not altered. However, this can only be done by manually feeding the grid attribute into the function grid_gen3D.to_graph_graphite() and collecting the object "idm" outputted by this function, then manually feeding the idm object and the grid and kpg attributes into graph_funcs3D.to_network(). This will **not** regenerate the kpn attribute, but will create an independent clone of the original kpn network object.

c. File output:
   i. Raw output datafile: Excel spreadsheet with layout described in the Data Analysis Methods section. The file will be deposited in the directory entered in bf_search3D.py, and be named excel_name+".xlsx".
   ii. Dissolution plot: Dissolution plot with layout described in the Data Analysis Methods section. The file will be deposited in the directory entered in dissolution_plot3D.py, and be named excel_name + "_plot.png".

Module: search_conductor

Performs multi-iteration 2D dissolution searches.

Functions:

1. triangular_search(rows, columns, boundary_sat,maxi_rad,maxi_amount,iterations,resolution, thresh): Performs multiple dissolution searches on triangular models. Each search will use a different model generated with the same parameters.
    a. Input:
        i. rows: Integer, number of rows in the grid.
        ii. columns: Integer, number of node stacks in each row.
        iii. boundary_sat: Integer, number of nodes in addition to the normal one that will be added to every boundary stack. Enter 0 if only one node per boundary stack is desired.
        iv. maxi_rad: Float, radius of maxicircle probability fields.
        v. maxi_amount: Integer, number of maxicircles to add to the model.
        vi. iterations: Integer, number of dissolution searches to run. Searches will be done in sequence, and a new model will be generated each time using the same parameters.
        vii. resolution: Integer, number of dissolutions in each dissolution step. For example, if resolution=3, then the program will delete three nodes from the model, then record component sizes to the output datafile, then delete three more, then record sizes, etc.
        viii. thresh: Integer, both the target component size during the dissolution search and the cutoff size for representation on dissolution plots. Each dissolution search will terminate when every component has fewer nodes than this, and only components of this size or larger will be plotted on the dissolution plot for each search.
    b. File output:
        i. Raw output datafiles: Raw output datafiles will be generated for each dissolution search, with the format described in the Data Analysis Methods section. Each will be deposited in the directory listed in bf_search.py, and be named [date]+'_RKP_NM_ triangular Grid_'+[iteration]. So, if a 3-iteration search were done on Sunday, August 16, the files would be named
        20200816_RKP_NM_ triangularGrid_1.xlsx,
        20200816_RKP_NM_triangularGrid_2.xlsx,
        20200816_RKP_NM_triangularGrid_3.xlsx.
        Obviously, you should change the initials to yours.
        ii. Dissolution plots: Dissolution plot with layout described in the Data Analysis Methods section. The file will be deposited in the directory entered in dissolution_plot.py, and be named

[date]+'_RKP_NM_triangularGrid_'+[iteration]+"_plot". So, if a 3-iteration search were done on Sunday, August 16, the files would be named 20200816_RKP_NM_triangularGrid_1_plot.png, 20200816_RKP_NM_triangularGrid_2_plot.png, 20200816_RKP_NM_triangularGrid_3_plot.png.

2. randomized_search(rad, rows, columns, boundary_sat, maxi_rad, maxi_amount, iterations,resolution,thresh ,reg_boundary=False,ringed=False): Performs multiple dissolution searches on randomized2D models. Each search will use a different model generated with the same parameters.
    a. Input:
        i. rad: Float, the radius of the probability fields of normal nodes.
        ii. rows: Integer, number of rows in the grid.
        iii. columns: Integer, number of columns in the grid.
        iv. boundary_sat: Integer, number of nodes in addition to the normal one that will be added to every boundary gridbox. Enter 0 if only one node per boundary gridbox is desired.
        v. maxi_rad: Float, radius of maxicircle probability fields.
        vi. maxi_amount: Integer, number of maxicircles to add to the model.
        vii. iterations: Integer, number of dissolution searches to run. Searches will be done in sequence, and a new model will be generated each time using the same parameters.
        viii. resolution: Integer, number of dissolutions in each dissolution step. For example, if resolution=3, then the program will delete three nodes from the model, then record component sizes to the output datafile, then delete three more, then record sizes, etc.
        ix. thresh: Integer, both the target component size during the dissolution search and the cutoff size for representation on dissolution plots. Each dissolution search will terminate when every component has fewer nodes than this, and only components of this size or larger will be plotted on the dissolution plot for each search.
        x. reg_boundary: Boolean, defaults to False. If True, any supersaturation of the model's boundary will be made regular. This has no effect if boundary_sat=0.
        xi. ringed: Boolean, defaults to False. If True, the model's boundary will be ringed, regardless of whether or not it is supersaturated. If ringed is set to True and reg_boundary is set to False, reg_boundary will be forcibly set to True as well.
    b. File output:
        i. Raw output datafiles: Raw output datafiles will be generated for each dissolution search, with the format described in the Data Analysis Methods section. Each will be deposited in the directory listed in bf_search.py, and be named [date]+'_RKP_NM_randomGrid_'+[iteration]. So, if a 3-iteration search were done on Sunday, August 16, the files would be named 20200816_RKP_NM_randomGrid_1.xlsx,

20200816_RKP_NM_randomGrid_2.xlsx,
20200816_RKP_NM_randomGrid_3.xlsx.
Obviously, you should change the initials to yours.

ii. Dissolution plots: Dissolution plot with layout described in the Data Analysis Methods section. The file will be deposited in the directory entered in dissolution_plot.py, and be named [date]+'_RKP_NM_randomGrid_'+[iteration]+"_plot". So, if a 3-iteration search were done on Sunday, August 16, the files would be named
20200816_RKP_NM_randomGrid_1_plot.png,
20200816_RKP_NM_randomGrid_2_plot.png,
20200816_RKP_NM_randomGrid_3_plot.png.

3. hexagonal_search(rows, columns, boundary_sat, maxi_rad, maxi_amount, iterations,resolution,thresh,with_pics=False,flag_bound=False): Performs multiple dissolution searches on hex2D models. Each search will use a different model generated with the same parameters.
    a. Input:
        i. rows: Integer, number of rows in the grid.
        ii. columns: Integer, number of columns in the grid.
        iii. boundary_sat: Integer, number of nodes in addition to the normal one that will be added to every boundary stack. Enter 0 if only one node per boundary stack is desired.
        iv. maxi_rad: Float, radius of maxicircle probability fields.
        v. maxi_amount: Integer, number of maxicircles to add to the model.
        vi. iterations: Integer, number of dissolution searches to run. Searches will be done in sequence, and a new model will be generated each time using the same parameters.
        vii. resolution: Integer, number of dissolutions in each dissolution step. For example, if resolution=3, then the program will delete three nodes from the model, then record component sizes to the output datafile, then delete three more, then record sizes, etc.
        viii. thresh: Integer, both the target component size during the dissolution search and the cutoff size for representation on dissolution plots. Each dissolution search will terminate when every component has fewer nodes than this, and only components of this size or larger will be plotted on the dissolution plots.
        ix. with_pics: Boolean, defaults to False. If True, component plots will be exported at the dissolution steps listed in hkinetoplast.dissolve_with_pics(). Currently incompatible with flag_bound. There is no reason why it has to be, I'm just fundamentally lazy. If both flag_bound and with_pics are set to True, with_pics will supersede flag_bound.
        x. flag_bound: Boolean, defaults to False. If True, each node will be flagged as either a boundary or non-boundary node, and the points at which the boundary develops its first and second holes will be noted in the output datafile and

dissolution plots. Currently incompatible with with_pics. If both flag_bound and with_pics are set to True, with_pics will supersede flag_bound.

b. File output:
    i. Raw output datafiles: Raw output datafiles will be generated for each dissolution search, with the format described in the Data Analysis Methods section. Each will be deposited in the directory listed in bf_search.py, and be named [date]+'_RKP_NM_hexagonalGrid_'+[iteration]. So, if a 3-iteration search were done on Sunday, August 16, the files would be named 20200816_RKP_NM_hexagonalGrid_1.xlsx, 20200816_RKP_NM_hexagonalGrid_2.xlsx, 20200816_RKP_NM_hexagonalGrid_3.xlsx.
    ii. Dissolution plots: Dissolution plot with layout described in the Data Analysis Methods section. The file will be deposited in the directory entered in dissolution_plot.py, and be named [date]+'_RKP_NM_hexagonalGrid_'+[iteration]+"_plot". So, if a 3-iteration search were done on Sunday, August 16, the files would be named 20200816_RKP_NM_hexagonalGrid_1_plot.png, 20200816_RKP_NM_hexagonalGrid_2_plot.png, 20200816_RKP_NM_hexagonalGrid_3_plot.png.
    iii. [Optional] Component plots: Component plot with layout described in the Data Analysis Methods section. For each dissolution search, a folder titled [date]+"_RKP_NM_hexagonalGrid_+[iteration]+["_base" OR "_maxi" OR "_sat" OR "_maxiSat"]+"_componentPlots" is created, where "_base" is used if there is no supersaturated boundary or maxicircles, "_maxi" if only maxicircles are used, "_sat" if only a supersaturated boundary is included, and "_maxiSat" if both a supersaturated boundary and maxicircles are included. Within each folder, the component plots will be named "after"+[number of elapsed dissolutions]+"dissolutions_componentPlot.png".

4. hex_xmax(xdim,spacing): Determines the maximum x-coordinate of a given hexagonal grid based on the number of columns and the spacing. Currently not used for anything,
    a. Input:
        i. xdim: Integer, number of columns in the hex2D model.
        ii. spacing: Float, distance between nodes in the underlying triangular lattices of the hex2D model.
    b. Output:
        i. x_max: Float, the maximum x-coordinate assigned to any node in a hex2D grid with the input parameters.
5. hex_ymax(ydim,spacing): Determines the maximum y-coordinate of a given hexagonal grid based on the number of rows and the spacing. Currently not used for anything,
    a. Input:
        i. ydim: Integer, number of rows in the hex2D model.

ii. spacing: Float, distance between nodes in the underlying triangular lattices of the hex2D model.

b. Output:

i. y_max: Float, the maximum y-coordinate assigned to any node in a hex2D grid with the input parameters.

6. rectangular_search(rows, columns, boundary_sat,maxi_rad,maxi_amount,iterations,resolution, thresh): Performs multiple dissolution searches on rectangular models. Each search will use a different model generated with the same parameters.

a. Input:

i. rows: Integer, number of rows in the grid.

ii. columns: Integer, number of columns in the grid.

iii. boundary_sat: Integer, number of nodes in addition to the normal one that will be added to every boundary stack. Enter 0 if only one node per boundary stack is desired.

iv. maxi_rad: Float, radius of maxicircle probability fields.

v. maxi_amount: Integer, number of maxicircles to add to the model.

vi. iterations: Integer, number of dissolution searches to run. Searches will be done in sequence, and a new model will be generated each time using the same parameters.

vii. resolution: Integer, number of dissolutions in each dissolution step. For example, if resolution=3, then the program will delete three nodes from the model, then record component sizes to the output datafile, then delete three more, then record sizes, etc.

viii. thresh: Integer, both the target component size during the dissolution search and the cutoff size for representation on dissolution plots. Each dissolution search will terminate when every component has fewer nodes than this, and only components of this size or larger will be plotted on the dissolution plot for each search.

b. File output:

i. Raw output datafiles: Raw output datafiles will be generated for each dissolution search, with the format described in the Data Analysis Methods section. Each will be deposited in the directory listed in bf_search.py, and be named [date]+'_RKP_NM_rectangularGrid_'+[iteration]. So, if a 3-iteration search were done on Sunday, August 16, the files would be named 20200816_RKP_NM_rectangularGrid_1.xlsx, 20200816_RKP_NM_rectangularGrid_2.xlsx, 20200816_RKP_NM_rectangularGrid_3.xlsx.

ii. Dissolution plots: Dissolution plot with layout described in the Data Analysis Methods section. The file will be deposited in the directory entered in dissolution_plot.py, and be named [date]+'_RKP_NM_rectangularGrid_'+[iteration]+"_plot". So, if a 3-iteration search were done on Sunday, August 16, the files would be named

20200816_RKP_NM_rectangularGrid_1_plot.png,
20200816_RKP_NM_rectangularGrid_2_plot.png,
20200816_RKP_NM_rectangularGrid_3_plot.png.

Module: search_conductor3D

Performs multi-iteration 3D dissolution searches.

Functions:

1. task(i): Does the actual model generation and dissolution search for randomized3D models. Each worker in the parallel pool generated by randomized_search() runs one iteration of this function.
    a. Input:
        i. i: Integer, the iteration index of the dissolution search. This will be used in the file names generated by the function.
    b. Hard-coded parameters: **To ease implementation of the joblib functions, all parameters for the search are hard-coded within task().**
        i. rad: Float, radius of each probability sphere for normal nodes.
        ii. rows: Integer, number of rows in each plane of the grid.
        iii. columns: Integer, number of columns in each plane of the grid.
        iv. planes: Integer, number of planes in the grid.
        v. boundary_sat: Integer, number nodes in addition to the usual one that will be placed in each boundary gridbox.
        vi. maxi_rad: Float, radius of the probability spheres of each maxisphere
        vii. maxi_amount: Integer, number of maxispheres.
        viii. resolution: Integer, number of dissolutions in each dissolution step. For example, if resolution=3, then the program will delete three nodes from the model, then record component sizes to the output datafile, then delete three more, then record sizes, etc.
        ix. thresh: Integer, both the target component size during the dissolution search and the cutoff size for representation on dissolution plots. Each dissolution search will terminate when every component has fewer nodes than this, and only components of this size or larger will be plotted on the dissolution plot for each search.
    c. File output:
        i. Raw output datafiles: Raw output datafiles will be generated for each dissolution search, with the format described in the Data Analysis Methods section. Each will be deposited in the directory listed in bf_search3D.py, and be named [date]+'_RKP_NM_randomGrid3D _'+[iteration]. So, if a 3-iteration search were done on Sunday, August 16, the files would be named 20200816_RKP_NM_randomGrid3D _1.xlsx, 20200816_RKP_NM_randomGrid3D _2.xlsx, 20200816_RKP_NM_randomGrid3D _3.xlsx.
        ii. Dissolution plots: Dissolution plot with layout described in the Data Analysis Methods section. The file will be deposited in the directory entered in

dissolution_plot3D.py, and be named [date]+'_RKP_NM_randomGrid3D_'+[iteration]+"_plot". So, if a 3-iteration search were done on Sunday, August 16, the files would be named 20200816_RKP_NM_randomGrid3D _1_plot.png, 20200816_RKP_NM_randomGrid3D _2_plot.png, 20200816_RKP_NM_randomGrid3D _3_plot.png.

2. randomized_search(iterations,threads): Performs multiple dissolution searches on randomized3D models. Each search will use a different model generated with the same parameters. Uses joblib.Parallel to allow multiple instances of the model to be processed simultaneously if given access to multiple CPU cores. Each worker in the pool will run an instance of task(i) with a different value for i.
    a. Input:
        iii. iterations: Integer, the number of dissolution searches to run. Searches will be done in sequence, and a new model will be generated each time using the same parameters.
        i. threads: Integer, the number of workers in the parallel pool. It should be noted that, despite the name, this is true parallel computation, not multithreading, e.g. there's no global interpreter lock shenanigans. I just suck at naming things.
    b. File output: See task().
3. graphite_search(xdim,ydim,zdim,sat,maxi_rad,maxi_amount,iterations,res,thresh): Performs multiple dissolution searches on hex3D models. Each search will use a different model generated with the same parameters. The probability sphere radius of normal nodes, used only for connecting to maxispheres, is hard-coded to be 1.
    a. Input:
        i. xdim: Integer, number of columns in each plane.
        ii. ydim: Intger, number of rows in each plane.
        iii. zdim: Integer, number of planes.
        iv. sat: Integer, number nodes in addition to the usual one that will be placed in each boundary stack.
        v. maxi_rad: Float, radius of the probability sphere of each maxisphere.
        vi. maxi_amount: Integer, number of maxispheres.
        vii. iterations: Integer, number of dissolution searches to run. Searches will be done in sequence, and a new model will be generated each time using the same parameters.
        viii. resolution: Integer, number of dissolutions in each dissolution step. For example, if resolution=3, then the program will delete three nodes from the model, then record component sizes to the output datafile, then delete three more, then record sizes, etc.
        ix. thresh: Integer, both the target component size during the dissolution search and the cutoff size for representation on dissolution plots. Each dissolution search will terminate when every component has fewer nodes than this, and only components of this size or larger will be plotted on the dissolution plots.

b. File output:
    i. Raw output datafiles: Raw output datafiles will be generated for each dissolution search, with the format described in the Data Analysis Methods section. Each will be deposited in the directory listed in bf_search3D.py, and be named [date]+'_RKP_NM_hexagonalGrid3D _'+[iteration]. So, if a 3-iteration search were done on Sunday, August 16, the files would be named 20200816_RKP_NM_hexagonalGrid3D _1.xlsx, 20200816_RKP_NM_hexagonalGrid3D _2.xlsx, 20200816_RKP_NM_hexagonalGrid3D _3.xlsx.
    ii. Dissolution plots: Dissolution plot with layout described in the Data Analysis Methods section. The file will be deposited in the directory entered in dissolution_plot3D.py, and be named [date]+'_RKP_NM_hexagonalGrid3D_'+[iteration]+"_plot". So, if a 3-iteration search were done on Sunday, August 16, the files would be named 20200816_RKP_NM_hexagonalGrid3D _1_plot.png, 20200816_RKP_NM_hexagonalGrid3D _2_plot.png, 20200816_RKP_NM_hexagonalGrid3D _3_plot.png.

Script: dissolution_plot_avg

Generates an averaged dissolution plot, with the structure described in the Data Analysis Methods section, from all Excel files within an indicated folder.

Functions:

1. excel_loader(path): Loads an excel file with the given name and directory.
    a. Input:
        i. path: String, full name and directory of the Excel file.
    b. Output:
        i. wb: xlrd Workbook object, the loaded Excel workbook.
        ii. ws: xlrd Worksheet object, the loaded Excel worksheet.
2. data_retriever(ws): Gathers the list of x-coordinates (dissolution steps) and y-coordinates (component sizes) from the loaded worksheet.
    a. Input:
        i. ws: xlrd Worksheet object, the loaded Excel worksheet. Must have the format of a raw output datafile, as described in the Data Analysis Methods section.
    b. Output:
        i. x_coords: Tuple of integers. All the dissolution steps as listed in the first column of ws.
        ii. y_coords: Tuple of tuples of integers. Each sub-tuple consists of the component sizes at a particular dissolution step. The nth element of the super-tuple is the list of component sizes associated with the nth element of x_coords.
3. plotter(x_coords, y_coords,export_name,io_name): Generates and saves an averaged dissolution plot using matplotlib.
    a. Input:
        i. x_coords: Tuple of integers. All the dissolution steps from the input Excel datafile with the most steps.
        ii. y_coords: Tuple of tuples of integers. The nth element is associated with the nth element of x_coords, and is the median list of component sizes across all the trials being averaged. See the Data Analysis Methods section for a description of how the median is determined.
        iii. export_name: String, (most of) the name of the output image file.
        iv. io_name: String, the title of the output plot (the title that will be printed on top of the plot in the image, not the file name).
    b. File output:
        i. Averaged dissolution plot: Averaged dissolution plot with layout described in the Data Analysis Methods section. The file will be deposited in the current working directory (which, if main() is run, will be the To_Aggregate folder at the directory entered in main()), and be named export_name+"_plot.png".

4. medianizer(x_master, y_master): Computes the median list of component sizes across all the trials being averaged at each dissolution step. See the Data Analysis Methods section for a description of the algorithm.
    a. Input:
        i. x_master: List of tuples of integers. Each tuple is the set of dissolution steps from a single trial.
        ii. y_master: List of tuples of tuples of integers. Each lowest-level tuple is a list of component sizes. Each mid-level tuple is the list of lists of component sizes for one of the trials being averaged. The top-level list is all the trials being averaged. The nth element of this list corresponds to the nth element of x_master.
    b. Output:
        i. x_coords: Tuple of integers. All the dissolution steps from the input Excel datafile with the most steps.
        ii. y_coords: Tuple of tuples of integers. The nth element is associated with the nth element of x_coords, and is the median list of component sizes at that dissolution step across all the trials being averaged. See the Data Analysis Methods section for a description of how the median is determined.
5. main(): This function should be run to have the program automatically invoke all the above functions in the proper order. The raw output datafiles from each trial you wish to average should be placed in a folder, and the path to the folder should be manually entered inside this function. Running this function will then output an averaged dissolution plot using all .xlsx files in the folder.

Script: delta_p_max_plot

Generates a P' plot, with format described in the Data Analysis Methods section, for each trial in a given directory. You may need to heavily modify all the hard-coded directories to match your machine/naming conventions.

Functions:

1. excel_retriever(direct): Loads an Excel file with the given name and directory.
    a. Input:
        i. direct: String, full name and directory of the Excel file.
    b. Output:
        i. wb: xlrd Workbook object, the loaded Excel workbook.
        ii. ws: xlrd Worksheet object, the loaded Excel worksheet.
2. data_retriever(ws): Gathers the list of x-coordinates (dissolution steps) and y-coordinates (component sizes) from the loaded worksheet.
    a. Input:
        i. ws: xlrd Worksheet object, the loaded Excel worksheet. Must have the format of a raw output datafile, as described in the Data Analysis Methods section.
    b. Output:
        i. x_coords: List of integers. All the dissolution steps as listed in the first column of ws.
        ii. y_coords: List of lists of integers. Each sub-list consists of the component sizes at a particular dissolution step. The nth element of the super-list is the set of component sizes associated with the nth element of x_coords.
3. p_prime(x,y): Computes the value P'(n) for each *n* in x using the largest elements of *y*.
    a. Input:
        i. x: List of integers. All the dissolution steps for a given trial.
        ii. y: List of lists of integers. Each sub-list consists of the component sizes at a particular dissolution step. The nth element of the super-list is the set of component sizes associated with the nth element of x
    b. Output:
        i. px: List of integers. A copy of x with the first element of x omitted.
        ii. py: List of integers. The nth element is the value P'(n).
4. plotter(x,y,save_loc,file): Generates and saves a P' plot using matplotlib.
    a. Input:
        i. x: List of integers. All the dissolution steps from the input Excel datafile with the first step omitted.
        ii. y: List of integers. The nth element is the value P'(n).
        iii. save_loc: String, (most of) the directory of the save folder.
        iv. file: String, the file name of the raw output datafile used to generate this plot, which will also be (most of) the name of the P' plot.

b. File output:
    i. P’ plot: P’ plot with layout described in the Data Analysis Methods section. The file will be deposited in one of four folders within the save_loc directory. If the substring “_base” is in file, then the image will be placed in the “BaseGrid” folder. If the substring “_sat” is in file, then the image will be placed in the “SuperSat” folder. If the substring “_maxi” is in file, then the image will be placed in the “Maxi” folder. If the substring “_maxiSat” is in file, then the image will be placed in the “MaxiSat” folder. The name of the image will be file+”_p_plot.png”.

5. gen_plot(direct,save_loc,file): Manages extraction of information from the raw output datafile and generation of the P’ plot for a single trial by calling the four above functions.
    a. Input:
        i. direct: String, full name and directory of the raw output datafile that will be used to generate the P’ plot.
        ii. save_loc: String, (most of) the directory of the save folder.
        iii. file: String, the file name of the raw output datafile used to generate the plot, which will also be (most of) the name of the P’ plot.
6. main(): This function should be run to have the program automatically invoke all the above functions in the proper order. Hard-coded in this function are the directory in which all the output datafiles live, as well as their sub-folders according to my personal naming and file hierarchy conventions. The function will dig through the folders, pick out those it has been told to use, and generate a P’ plot for every trial within them.

Script: delta_p_max_plot_avg

Generates an averaged P' plot, with format described in the Data Analysis Methods section, using all trials in a particular folder.

Functions:

1. excel_retriever(direct): Loads an Excel file with the given name and directory.
    a. Input:
        i. direct: String, full name and directory of the Excel file.
    b. Output:
        i. wb: xlrd Workbook object, the loaded Excel workbook.
        ii. ws: xlrd Worksheet object, the loaded Excel worksheet.
2. data_retriever(ws): Gathers the list of x-coordinates (dissolution steps) and y-coordinates (component sizes) from the loaded worksheet.
    a. Input:
        i. ws: xlrd Worksheet object, the loaded Excel worksheet. Must have the format of a raw output datafile, as described in the Data Analysis Methods section.
    b. Output:
        i. x_coords: List of integers. All the dissolution steps as listed in the first column of ws.
        ii. y_coords: List of lists of integers. Each sub-list consists of the component sizes at a particular dissolution step. The nth element of the super-list is the set of component sizes associated with the nth element of x_coords.
3. medianizer(x_master, y_master): Computes the median list of component sizes across all the trials being averaged at each dissolution step in x_master. See the Data Analysis Methods section for a description of the algorithm.
    a. Input:
        i. x_master: List of lists of integers. Each sub-list is the set of dissolution steps from a single trial. The super-list contains one such set from each trial being averaged.
        ii. y_master: List of lists of lists of integers. Each lowest-level list is a list of component sizes. Each mid-level list is the list of lists of component sizes for one of the trials being averaged. The top-level list is all the trials being averaged. The nth element of this list corresponds to the nth element of x_master.
    b. Output:
        i. x_coords: List of integers. All the dissolution steps from the input Excel datafile with the most steps.
        ii. y_coords: List of integers. The nth element is associated with the nth element of x_coords, and is the median maximum component size at that dissolution step across all the trials being averaged. See the Data Analysis Methods section for a description of how the median is determined.

4. p_prime(x,y): Computes the value P'(n) for each *n* in x using the largest elements of *y*.
    a. Input:
        i. x: List of integers. All the dissolution steps for a given trial.
        ii. y: List of integers. The nth element is associated with the nth element of x_coords and is the median maximum component size at that dissolution step across all the trials being averaged.
    b. Output:
        i. px: List of integers. A copy of x with the first element of x omitted.
        ii. py: List of integers. The nth element is the value P'(n).
5. plotter(x,y,save_loc,title): Generates and saves an averaged P' plot using matplotlib.
    a. Input:
        i. x: List of integers. All the dissolution steps from the input Excel datafile of the trial with the most dissolution steps, with the first step omitted.
        ii. y: List of integers. The nth element is the value P'(n) using the medianized list of maximum component sizes.
        iii. save_loc: String, the directory of the output image file.
        iv. title: String, the title of the output plot (the title that will be printed on top of the plot in the image) and part of the file name.
    b. File output:
        i. Averaged P' plot: Averaged P' plot with layout described in the Data Analysis Methods section. The file will be deposited in the save_loc directory with name title+"_p_plot.png"
6. main(): This function should be run to have the program automatically invoke all the above functions in the proper order. Hard-coded in this function are the directories of the input and export files, as well as the name of the output file. When invoked, the program will run through every Excel file in the given directory, extract the relevant information, and combine it into an averaged P' plot which will be saved to the requested directory.

Known Limitations

Lack of Probability Function Orientability

Both probability circles and spheres are given uniform orientation. Probability functions are defined in polar coordinates for circles and cylindrical coordinates for spheres. Therefore, they are defined relative to fixed coordinate axes. The program currently does not allow the orientation of these reference axes to be randomized; the axes of all nodes' fields all point in the same direction. Thus, while angular dependence in the probability function is possible, the $\theta=0$ axis of all circles and spheres will always be parallel. This also effectively means ellipses, polygons, or any other shapes without radial symmetry cannot be used for probability fields.

Semi-Manual Integration for 2D Probability Functions

Angular dependence in 2D probability fields is further hampered by Python's bizarre inability to run one very particular kind of radial integral. In the very specific case where the probability function of a larger circle must be integrated over the area of a smaller circle that is entirely within the larger circle, scipy has a tantrum. The results of the radial integral in such a situation will have a massive error attached to them. Angular integral, fine. But no radial integrating for you. When I run the exact same integral by hand, or when I have Mathematica run it numerically, we get the same correct value. There is always the very real possibility that I'm just a bit dumb and managed to mistype the Python commands three times in a row and not see it, but I could not figure out why integration fails in this very specific case.

So, in this case, and this case only, Python will only integrate the angular component of the probability function, and you have to manually integrate the radial component of whatever probability function you're using and enter it in prob_funcs.prob_integrator(). The integral currently sitting in the function is for the probability function currently entered in prob_funcs.prob_func(). It should be noted, however, that the only time this matters is if you anticipate having to integrate probability functions of different sizes, e.g. when maxicircle link probability is determined by probability function integration. **Currently, this situation never occurs, so this is not a problem that ever arises with the current model.**

To be clear, in every other situation (see Fig 5.1) the double integrals work fine, and Python will do all the integrating for you. In 3D, the integrals work in every situation. Only in the one, specific situation for 2D fields does it fail. This problem will only need to be addressed if you change the modeling algorithm so that it becomes relevant.

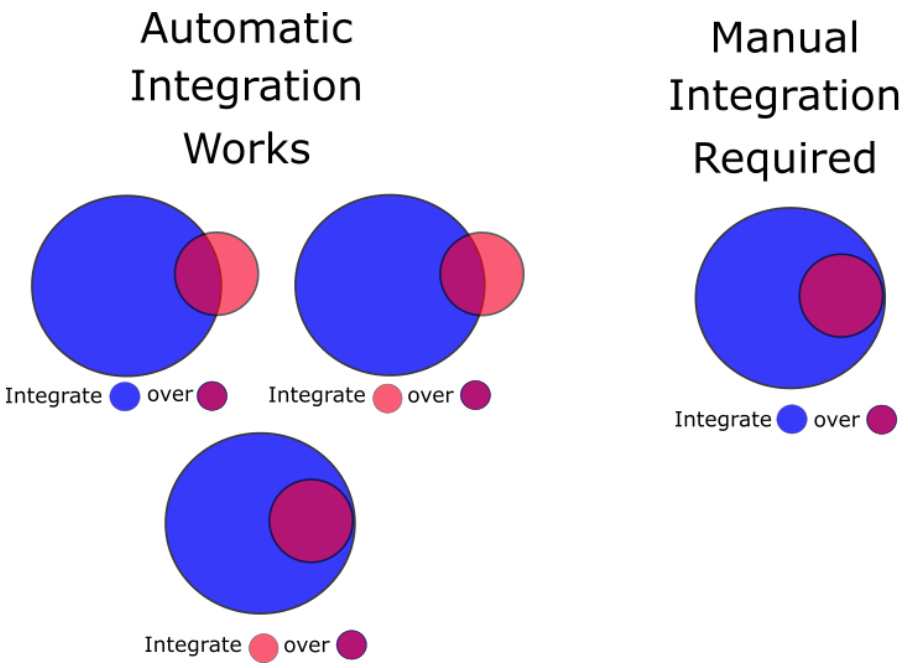

Figure 5.1: Situations when scipy's numerical integrator works and the one situation in which it does not. Currently, the situation at right never occurs. All probability fields of normal nodes have the same radius and therefore cannot entirely envelope one another. Maxicircles automatically link to any nodes with fields intersecting their boundaries, so even though they can entirely envelope other probability functions, no integrals are performed. If this is all still very confusing for you, take a look at the decision tree and inline comments in prob_funcs.prob_integrator() to see what's actually going on in the code.

Hard-Coded Directories

This has been mentioned throughout this document, but all directories currently point to locations on my computer, and can only be changed by manually editing the code. This is very convenient for me, but very inconvenient for everyone else. You may get several "file not found" errors before you manage to track down everywhere there's a hard-coded directory. Sorry.

Exceeding Trial Length in Averaged Dissolution Plot Algorithm

The current algorithm in dissolution_plot_avg.py deals with having to average multiple trials that terminate at different x-values by ignoring trials whose lengths have been exceeded. For example, say four trials are being averaged. Trial 1 terminated after 420 dissolution steps, Trial 2 after 430 dissolution steps, Trial 3 after 440 dissolution steps, and Trial 4 after 451 dissolution steps. The median across all four trials would be computed for the first 420 x-values. On the 421st dissolution step, however, Trial 1 has no entry. So, it is simply removed at this point. Between x=421 and x=430, the median of Trials 2, 3, and 4 is used. Then Trial 2 is deleted, so the median of Trials 3 and 4 is used until x=441, at which point all remaining y-coordinates will be drawn solely from Trial 4.

This is in contrast to the averaged P’ plot algorithm, which, rather than ignoring trials whose length has been exceeded, instead assigns them a y-value of 0 at all further x-values. With this algorithm, when x=421 in the above example, Trial 1 is not removed, but instead assigned a y-value of 0, and the median of 0 and the max component size of Trials 2, 3, and 4 is used. The averaged P’ plot algorithm was changed this way in order to avoid negative values of P’.

Whether this is a problem or a feature depends upon how you want to do your medianizing. Both algorithms are valid, though I prefer the latter as it ensures a monotonically decreasing maximum component size in the averaged plots.