Samuel Chicoine
8/2/2024

Drone Simulation Development

This program is based on the research paper “Moving Aerial Base Station Networks: A Stochastic Geometry Analysis and Design Perspective” (Enayati et al. 2019). This program implements three of the trajectories covered, simple radial, ring and spiral. These trajectories are designed so that drones following them will evenly cover the circular area they traverse. This development guide will not cover the proofs or derivations of these trajectories but instead how they were implemented in this program. This guide will work through each function of each file used for this program starting with main, then drone and lastly sub area .

First lets cover some basic concepts for the simulation, the simulation attempts to measure the coverage the circle receives to show that the trajectories do in fact provide uniform coverage given enough drones. To do this we must be able to measure which part of the circle a given drone is covering. Splitting up a circle into even sub regions leaves us with little options. It was decided that tiling with squares would be easy to work with and be increasingly accurate the smaller the sub area square size. Thus the basic premise is to create a circular area, tile it with square sub areas, and measure which sub areas receive coverage as the drones complete their trajectories.

I also want to cover some of the initial choices made before beginning development, but if you don’t care feel free to skip. First to address is, in my opinion, the elephant in the room which is why would something like this be developed in Python. I knew from the start Python would have its performance limits and I wasn’t enthusiastic about working with the Tkinter library. I spent most of my first week and a half, searching for what programs might be best to create such simulations, and I came up very empty handed. The entire time I was remembering how I had used Python with the Tkinter library to create Space Invaders. The two might not seem super related but the ability to manipulate objects on canvas and check for “collisions” as I had done before kept this language library combo in the back of my mind until I finally caved so I could begin developing the program.

Main.py

To start, let's cover the libraries. Tkinter is the standard GUI library for Python. Tkinter allows us to create a window with a canvas and then draw and move shapes on this canvas. Tkinter does not keep track of each object's coordinates for us so if we wish to manipulate shapes in meaningful ways we must keep track of the coordinates ourselves. Next should be a more familiar library numpy, this should be pretty self explanatory but we will undoubtedly need quite a few mathematical functions from this library, as well as our next library expecting data in numpy array form. The next library of course being matplotlib specifically the pyplot interface, this allows us to visualize our results for the user at the end of each simulation. We also import the drone.py and sub_area.py dependencies which will be covered later.

main

Let's start with the main function, main handles taking user inputs, handing these to our run simulation function then graphing the results from run simulations return. Then we handle a couple more user inputs for if the user wishes to save the results graph and/or run another simulation. To do this we start by declaring all of the user inputs each time we enter our "again" loop, which determines whether or not to run the program again. Our simulation relevant user inputs include the radius of circle the drones will cover, the trajectory the drones will use, the drone's own coverage radius, the number of drones, the side length of the sub areas and the number of repetitions of the trajectories to complete.

Now declaring them all as -1 is somewhat arbitrary but it allows better verification of user inputs as then we place each input into a loop until a valid input has been entered and the value changes. The use of try excepts here is not my proudest programming, particularly because of how this interacts with actual interrupts but this seems to be by far the easiest way of validating user inputs.

Note that the square side input is where we see our first bit of scaling. Early in development everything changed size just as the user inputted. However I quickly realized that it would be much better to keep the visual size of the circular area the same and scale everything else accordingly. Our circle exists as a 200 pixel radius circle in a 600 by 600 canvas with a center of 300x300. If the user inputs a radius of 100 the sub areas and drones will scale up by a factor of two while keeping the visual size of the circular area the same. This keeps the simulation consistent while preventing simulations from having hard restrictions on user inputs to prevent things from drawing outside the window.

This is also where I should introduce essentially our accuracy limit that being of a single pixel, or unit. This starts with the fact that our circular area is not in fact circular but an approximation using pixels. The pixels exist as 1x1 squares for our purposes and although calculations can certainly be done to a much greater degree of accuracy, our circular areas exist as a construct of these 1x1 squares. Thus sub areas of 1x1 size would in fact tile our circular area as accurately as possible meaning sub areas smaller than a single unit are mostly meaningless in terms of improving accuracy. However because of the way scaling works if the user inputs a radius greater than the actual 200 pixel radius we would get a scaling factor of less than 1. Then even if the user inputs a sub area side length greater than one it could end up less than one because of scaling. So we include the max() statement to ensure the sub area side length does not become less than a single pixel unit.

Once we have gathered relevant simulation inputs we call the run simulation function which will return a list of the coverage values each sub area has received. Once we have this data we need to graph it. This is very basic matplotlib usage, the sub area IDs simply correspond to their index in the sub area list which can just be recreated using the linspace function. Then just plot our two lists of data and as a bar graph and show the graph. Then we want to handle the last couple user inputs which is if the user wants to save the graph results which will become a png. And then if the user wants to run the simulation again.

run_simulation

This begins with setting up our visuals. First we must declare a Tkinter root which essentially is our window that will contain all our visual aspects. However to create shapes onto this window we must declare a canvas on it which we simply call canvas. Then we will create the first object on this canvas, our circular area the drones will cover. Now remember the visual size of this circle never changes so it is always declared simply based on the canvas size. Then we need to figure out the scaling factor to use for the other objects created. This is simply the scale difference between the user input radius and the actual radius of 200 pixels or canvas_size/3. If the user inputs a radius of 100, we will get a scaling factor of $\frac{600}{3} \cdot 100$ or two which means which will create everything else twice as large as their given value. We also use

Tkinter to bind our escape key to a function which will be covered later but it is used for stopping the simulation early if the user presses escape.

Now it is time to create our sub areas, the way we break our circular area up into regions so we can actually measure what part of the area is being covered at any given moment. We simply call the function that performs this which we will cover later, for now it just returns a list of all the sub areas objects. Lastly, after a couple of tkinter calls, we “pack” the canvas which places it onto the window and then update the canvas which loads all the changes we have made at once.

Then we will create our drones, first we create our list for our drones based on the number of drones the user inputted. Then we just enter a loop to fill this list with drone objects passing in the relevant parameters. We will cover the drone class later.

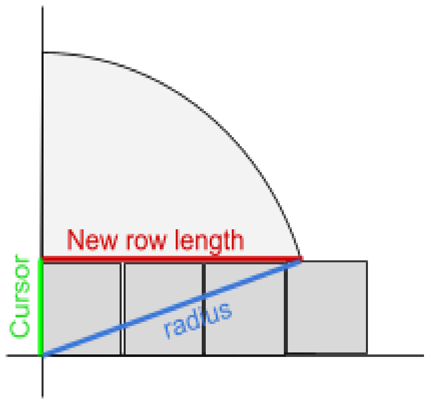
Now it is time to enter our simulation loop which will continuously call updates until all drones are finished. These updates themselves are really just the relationship between how often the visuals should update and the amount the drones have actually changed. The drones themselves never make a move more than one unit at a time. This is because as previously defined our sub areas will never be smaller than a single unit. If we move more than one unit at a time we may skip over sub areas. How much less than one unit we move our drone doesn’t really matter as long as we scale coverage accordingly which you will see in how we handle decimal velocities.

It also complicates things that the different trajectories differ in how often the visuals update. We will get in depth into this later but this while loop will run until all drones are finished as indicated by their self.finished attribute. The loop will end early if the global exit_simulation flag is set to true. In the loop we first start by measuring the time it is going to take for the update to complete. Simply for user purposes we don’t want the simulation running as fast as possible or else trivial simulations will hardly appear before completing. We take the duration of the update and if it is less than 0.01 seconds we will sleep the difference. Of course the program can still slow down if updates start taking longer than 0.01 seconds.

Once every drone is finished and we break out of the loop, put our coverage results into a list and return it. We could also just return the sub area list but the actual coverage values need to be extracted either way and because the sub area IDs just correspond to their index we can just get the coverage data and return it.

generate_sub_areas

Finally we get into our real first algorithm which is tiling our circle with square sub areas. The basic premise here is to start in the upper right quadrant, then go row by row creating sub areas until that row is covered, then move up to the next row. Let's start with the first row, the length of this row will simply be the radius of the circle. Then a ceiling divide of this row length by the sub area side length will tell us how many sub areas it will take us to cover this row. A Python trick to ceiling divide is using -(x // -y) on two numbers x ceiling divided by y. To create each sub area we need two coordinate pairs specifying the top left and bottom right of each square handed to the sub area class constructor which will be covered later.

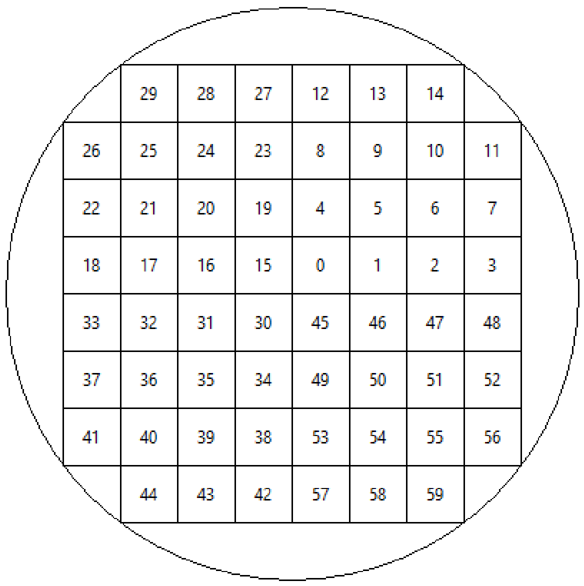


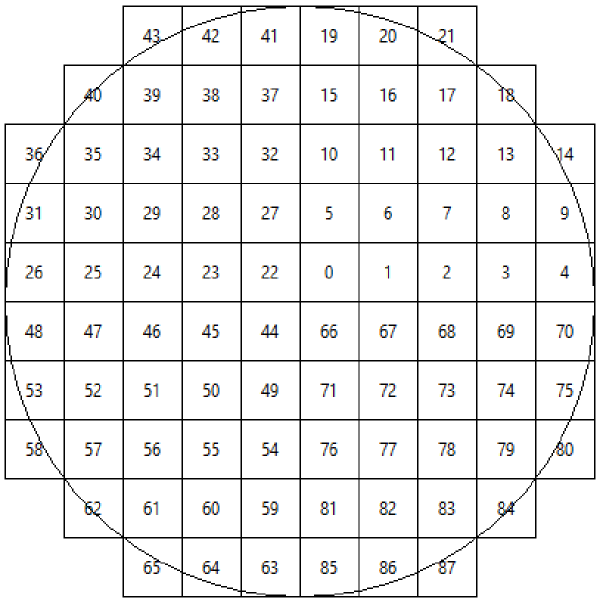
We will increment these coordinates horizontally by the sub area side length each row and vertically by the cursor for each new row. We use the cursor variable to keep track of which row we are on incrementing it by the side length of a sub area until all rows are complete. However the length of each row is not constant, at first it is the radius but each new row requires us to solve a simple pythagoras theorem to get the next row length as depicted.

Although this process is straightforward enough to apply to the other quadrants there is a much easier way by the fact that the quadrants are symmetrical. Instead of repeating the entire process four times instead each time we create a sub area we create three additional sub areas with flipped coordinates. However this would then make our list of sub areas disorganized for our purposes. This is why we first calculate the number of sub areas per quadrant then we place sub areas into the list based on which quadrant it will belong to to keep the sub areas in order.

generate_sub_areas_less

This is another version of generate_sub_areas with the difference being this will only create sub areas that completely fit into the circle avoiding sub areas that get cutoff.





The difference is shown above when using a circular area of 100 and sub area side length of 20. As expected any areas that would be cut off do not get created. The way this works in the program really just means changing the ceiling divides to floor divides. The functions are extremely similar otherwise.

drone.py

The drone python file contains everything for the drone class and all relevant functions. This is the largest file and is where all calculations to move drones and calculate coverage happens. The libraries here should be self explanatory, we will need the tkinter library to make adjustments to our canvas as well as numpy for a lot of the calculations, and as seen later we do need some random values so that is imported. For the program we start with our drone class, we will take full advantage of OOP here, defining all the attributes our drone will need and creating functions to easily manipulate these drone objects. Starting with our constructor which creates a drone object given the aforementioned parameters and some specific parameters depending on which trajectory the drone object is going to follow. We also have the drones active and finished flags, a drone must be not finished and active in order to receive updates. Not all drones start active which is why there is a differentiation. We also declare an attribute R to be the drone's distance from the center of the circular area. And lastly we create a corresponding circle on our canvas representing the drone.

next_radial

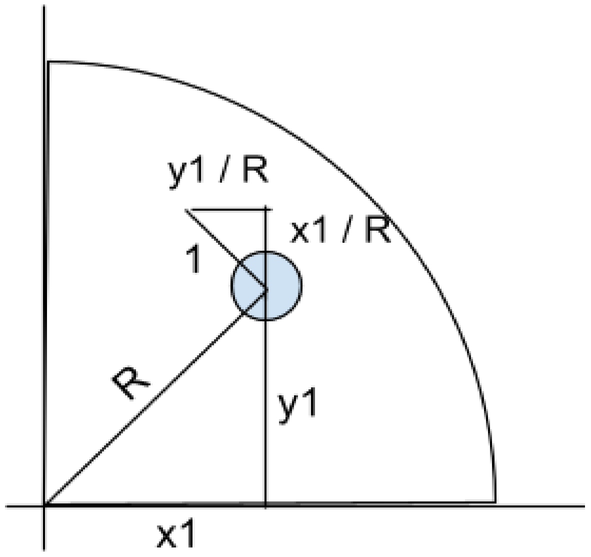
Now is a good time to introduce the trajectories briefly, detailed proofs and derivations are in the original research paper “Moving Aerial Base Station Networks: A Stochastic Geometry Analysis and Design Perspective” (Enayati et al. 2019). The first trajectory we will go over is the radial trajectory. First the drones are angled out evenly across the entire range of 0 to 2pi. Then the basic premise is that the drones will travel at their respective angles outwards, reach the area's edge then travel back completing one cycle. Although this evenly spreads out the drones this would not result in even coverage. The drones are far more clustered at the center and spread out as they travel. Thus the velocity of the drone is made dependent on the distance from the center slowing down as the drone gets farther. The exact equation for the velocity is $v = \frac{p^2}{R \cdot \tau}$, where p is the area’s radius, R is the drones distance from the center and τ is an arbitrary constant used to inversely scale velocity. This will account for the way the drones cluster towards the center resulting in even coverage across the circular area.

As mentioned before we do not want to move the drone more than one unit at a time. The only question then is how many moves do we want to make in between updating our drone visually. Remember in our run simulation function we update the canvas after each call of each drone’s respective next trajectory function.

Well because our radial trajectory uses a continuously changing velocity this gets a little tricky. First it is important to note that the frequency of visual updates are arbitrary, all relevant coordinates and coverage are calculated separately and the visuals act as a thin layer on top. That being said, the way I approached this was a bit backwards, each radial update will move a drone until it has completed one unit of coverage worth of movement before returning and allowing our run simulation to update the visual. If this is confusing perhaps read the more straightforward ring trajectory function and come back. Now for each unit we move we will calculate the velocity and add coverage inversely proportional to the velocity to the sub areas covered. The higher the velocity the less time a drone will be covering an area. Because movements of one unit will not usually evenly add up to one unit of coverage we check and see if another addition of one unit's worth of coverage will go past and if so instead calculate a move with distance based on the remaining coverage amount. Again we just have to avoid movements greater than one unit at a time. We also need to include checks to turn the drone around once it has reached the edge which is programmed as the flip attribute. We simply flip the changes to our drones x, y and corresponding circle on the canvas if this attribute is true. We also need to check if the drone has reached the center again. This is easily checked by seeing if the drones flip attribute is true and if it's within one unit of the center. Then that is one cycle complete and if all cycles are complete then we can remove the drone setting it as not active, finished and deleting the corresponding circle on our canvas.

next_ring

The ring trajectories premise is that we can create even coverage of a circling using orbiting drones. The drones will all have the same orbit period meaning farther out drones travel faster. Then to create even coverage the orbit radius is weighted so that drones will be more clustered farther out offsetting the reduced coverage of a higher velocity resulting in even coverage. The orbit radius is calculated as $r_i = \sqrt{\frac{i}{N+1}}p$ where i is the ith drone accordingly to its ID/ index in the drone list, N is the total number of drones and p is the circular area's radius as before. The velocity is simply $v_i = \frac{2\pi r_i}{\tau}$ where r_i is the ith drones orbit radius and τ is an arbitrary constant so that each drone will orbit every τ units of time, depends on the frequency of updates. Because each drone has a constant velocity we can simply move the distance of each drone's velocity each update. Again we are going to break this down into moves of one unit and then a remainder if necessary. Then unit by unit we work through the velocity checking the sub areas covered at each with again coverage being inversely proportional to velocity.



To actually orbit we use the simple method of "continuously" moving perpendicular to the line formed between the drones center and the center of the circular area. To move along this tangent we can flip the adjacent and opposite sides of the triangle formed with the center of the circular area, the drones center and the x-axis. Then because we only want to move one unit we simply scale that new “perpendicular” triangle so that its hypotenuse is one unit resulting in a scaled and flipped ratio of x and y for us to move as seen in the diagram.

We will move $y1 \div R$ in the x direction and $x1 \div R$ in the y direction to achieve a movement of one unit along this tangent. However, whether to add or subtract these values is dependent on which quadrant we are in as well as the direction of orbit we want. In this program counter clockwise rotation was chosen. Thus we have added three conditions for each move in x and y. If our drone’s distance in x from the center is greater than 0 then we are in quadrants one or four (the right two quadrants), thus we should adjust y upwards. If the distance is equal to 0 (the top and bottoms of the orbit) then there will be no shift in y. Otherwise the distance in x is less than 0 then we want to adjust our y downwards. Do note that because of tkinter our y axis is flipped. (0,0) is the top left coordinate and moving downwards means increasing the y value. A very similar check is done for the x move, if the distance in y is greater than 0 (bottom two quadrants) then we should increase our x moving it to the right. If the distance in y is 0 then x won't change at all. And if the distance in y is less than 0 then we will decrease our x moving it left.

Now this algorithm for orbiting works great in theory and almost in the program.You see thanks to floating point errors the drones slowly drift off course. However remember that we initially calculated at which radius each drone should orbit. And by comparing the intended orbit radius to the current drone’s distance from the center we can obtain a factor of how far off the drone is from its intended orbit. Then we can simply use this factor to scale the drones position back into orbit. This error correction is done once per update at the end of the function. It is possible to notice a slight wobble in the drone's orbit but that is preferable over the drone simply drifting off.

Lastly in the ring trajectory a cycle is defined as one complete orbit, thus each time we complete an orbit we want to decrement our cycles until all cycles are complete. If so then we need to set our drone as inactive, finished and delete the corresponding circle on our canvas.

next_spiral

This is definitely a change of pace, the other two trajectories followed paths that are intuitive and easy mathematically to implement. Basically the functions involved in solving for coordinates are pretty trivial. However the spiral trajectory is very function heavy and requires many derivations to implement. Before beginning please note that I used desmos extensively to test and help myself visualize the functions used for this trajectory which can be found at this link. www.desmos.com/calculator/lew2tky16w.

Let's first start with the trajectory itself, assuming you are familiar with polar coordinates the most simple polar spiral is the archimedes spiral simply given by $r = \theta$, in cartesian this can be written as $f(x) = (x\cos(x),\ x\sin(x))$. We will be using a variation of a polar spiral defined in the paper as $X(s) = (ps^k\cos(\zeta s),\ ps^k\sin(\zeta s),\ s \in [0, 1]$. Where p is our area's radius, k is a constant affecting the size of the spiral and ζ determines what angles the function will cover, as it is only defined for values 0 to 1. For our purposes we will choose values given in the paper's example except for area radius which will be generated based on user input. This gives us the equation $X(s) = (ps^2\cos(2\pi s),\ ps^2\sin(2\pi s),\ s \in [0, 1]$, but this isn't quite ready. This equation functions so that X(s) will complete angle ζ for $s \in [0, 1]$, this just isn't a level of control we care about for our program so instead we will use the equivalent $X(\theta) = (\frac{p\theta^2}{4\pi^2}\cos(\theta),\ \frac{p\theta^2}{4\pi^2}\sin(\theta))$, this function follows the exact same spiral but is directly in terms of our angle $\theta \in [0, 2\pi]$. The polar function of this is $r = \frac{p\theta^2}{4\pi^2}$, using these will make our work onwards much simpler.

As before we need to avoid making moves greater than a single unit at once. If we move one unit of arc length along the spiral it is impossible for it to be a move greater than a single unit. We then need a function to determine the arc length of a spiral function in terms of θ. The general formula for this is given by $\int_{\theta_1}^{\theta_2}\sqrt{r^2 + (\frac{dr}{d\theta})^2}\,d\theta$, where r is our polar spiral function $r = \frac{p\theta^2}{4\pi^2}$, and thus $\frac{dr}{d\theta} = \frac{2p\theta}{4\pi^2}$. Plugging in our gets us $\int_{\theta_1}^{\theta_2}\sqrt{(\frac{p\theta^2}{4\pi^2})^2 + (\frac{2p\theta}{4\pi^2})^2}\,d\theta$, this can be simplified to $\frac{p}{4\pi^2}\int_{\theta_1}^{\theta_2}\theta\sqrt{\theta^2 + 4}\,d\theta$, then we will perform u substitution with $u = \theta^2 + 4$, and thus $\frac{du}{d\theta} = 2\theta$. This gives us $\frac{p}{8\pi^2}\int_{\theta_1}^{\theta_2} \sqrt{u}\, du$, then solving the integral gets us $\frac{p}{8\pi^2}\,(\frac{2}{3}u^{\frac{3}{2}})$, replace u, finally getting us $\frac{p}{8\pi^2}\,(\frac{2}{3}(\theta^2 + 4)^{\frac{3}{2}})$, evaluated from θ_1 to θ_2. Now for our purposes we will always evaluate arc length starting at 0 to some angle θ, plugging this in as our evaluation points gets us $\frac{p}{8\pi^2}\,(\frac{2}{3}(\theta^2 + 4)^{\frac{3}{2}}) - \frac{p}{8\pi^2}\,(\frac{2}{3}(0^2 + 4)^{\frac{3}{2}})$, now we just simplify to finally get $L(\theta) = \frac{p}{12\pi^2}\,((\theta^2 + 4)^{\frac{3}{2}} - 8)$, which is the arc length from 0 to angle θ of our spiral function $r = \frac{p\theta^2}{4\pi^2}$.

Now remember that we want to move in specific increments no greater than one unit along this arc length. In other words we want to be able to give an arc length and get a corresponding angle to use in our cartesian function to solve for the corresponding x and y values. In other words we solved for arc length in terms of angle but we want angle in terms of arc length. This is of course very straightforward simply rearrange our equation to get $A(l) = \sqrt{(\frac{12\pi^2 l + 8p}{p})^{\frac{2}{3}} - 4}$, for $l \geq 0$, l is given arc length. Please note that this equation does not work for negative arc lengths, luckily for us we will not be moving negative arc lengths so this equation works in the defined range.

But that is just half of the puzzle, just like radial our drones velocity is not constant. Intuitively this makes sense as just like in radial our drones will be more clustered around the center thus they should move slower as they get farther out. Sadly obtaining this velocity is not nearly as straightforward, in fact following the paper we somewhat work backwards. The paper provides the means to obtain the x and y coordinates a drone should be at by a given time. This can be fairly easily translated as velocity, move this amount by this time. However we need to go through a few steps first. We will start with a value $\tau > 0$, as you will soon see this actually serves as the time it will take for the drone to complete its trajectory, functioning as a value to control the drones speed. Now the paper gives a more complicated basis for the timings of the drones, we start with T_i an independently uniformly chosen value from $(0, \tau)$ corresponding to the ith drone. Then we define each ith drone to move when $k\tau + T_i \leq t \leq (k + 1)\tau + T_i$. Note this has the interval length of τ, which we will show more clearly in a moment.

First let's actually introduce the formula for x and y in terms of time, $(x_i(t), y_i(t) = (ROT_{\theta_i}(x_i(h^{-1}(t - k\tau - T_i), y_i(h^{-1}(t - k\tau - T_i))$, first thing you will notice is ROT_{θ_i}, this is an independently chosen random value from $(0, 2\pi)$ which serves as the random rotation each drone will use to evenly spread them out across the area's angle. Now take note that we don't actually plug in the raw time value we plug in $t - k\tau - T_i$, this is important as going back to the ith drone's time range and solving for $t - k\tau - T_i$ instead of t, we see that the actual values we plug into the equation will be in the range $0 \leq t - k\tau - T_i \leq \tau$. This of course makes sense and lines up with the fact that our drone will take τ time to complete. Although I'm not sure if this is the exact right wording I like to think of this $t - k\tau - T_i$ as normalizing or shifting the time inputs to simply go from 0 to τ.

However the real important factor here is the h^{-1} function. Note that our equations for x and y in terms of time are still just the x and y equations. What is different here is that we apply this h^{-1} function to our time inputs and that is what makes the equations work in terms of time. I like to think of this h^{-1} function as a translation from time values to theta values. Plugging our $t - k\tau - T_i$ into $h^{-1}(t - k\tau - T_i)$, gets the corresponding angle that can be then plugged into our cartesian equations for the x and y coordinates corresponding to that time value. Then the intuitive goal of this h^{-1} function will be to make increments at lower values of time correspond to greater outputs resulting in a faster movement along our spiral at lower times and slower movement as t approaches τ.

The paper gives us $h(s) = \frac{\tau r(s)^2}{p^2}$, where $r(s) = \sqrt{x^2 + y^2}$, or simply this distance from the drone's center to the origin, in our case the center of our circular area. Solving for the inverse gets us $h^{-1}(s) = 2\pi\sqrt[4]{\frac{r(s)}{\tau}}$. Now we can finally discuss applying this into the program, we get our time value based on the number of ticks and length of each tick as defined before in the program's main file. We have to adjust this time value back every cycle as well as by the

$- k\tau - T_i$. Now we hand this into our next_spiral function that will handle updating the drone's spiral trajectory for each tick. The first step is of course plugging in the time value into our h^{-1} function, as stated this translates our time value into a corresponding value of θ. As we keep track our drone's current θ starting at 0 we can calculate the current arc length and the arc length for this update's new time value. Now we simply subtract the two and this is the distance along the arc length we need to move for this update. Now as before can can break this down into movements of one arc length at a time, we increment our current arc length by one and use this in our equation for θ in terms of arc length to get corresponding values of θ for each increment of arc length. Then each time simply plug this new θ into our cartesian equations for x and y and move the drone accordingly updating our drone's x and y coordinates. If we reach a remainder we know this will be our last move so we simply set our current arc length to the earlier calculated length for the update's time value. We use this to calculate a corresponding theta and move and we scale the coverage inversely proportional to the size of the move.

check_covered_areas

This function handles looping through our sub area list using the indexes specified from the get_CCA_indexes function and calling the check_overlap function to determine which sub areas are within the drone's coverage radius and if so adding to that sub area's coverage. The get_CCA_indexes function returns either one or two pairs of coordinates so the outer loop will loop for each pair and the inner loop will use each pair of indexes to index our sub area list.

get_CCA_indexes

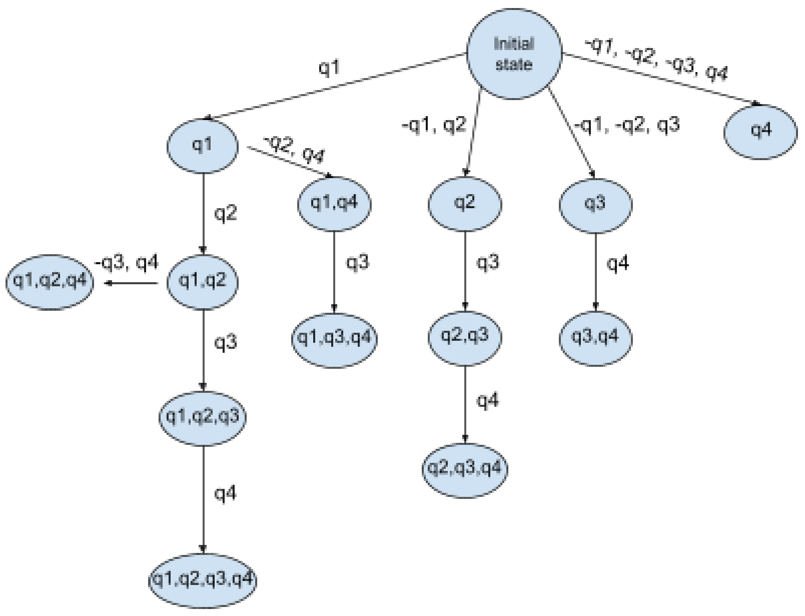
To first understand the point of this function, first consider a base scenario. We have moved a drone and want to check what areas it is covering. We can simply loop through every area calling check_overlap on our drone with each sub area until all have been checked. This works but is frankly quite inefficient. There should be a way to narrow down which sub areas we check based on the drone's general location. And thus the motivation for this function came to be.

First we start with the fact that our sub area list has been organized so that each quarter of the list corresponds to the sub areas of each quadrant. In reality this was something I had to implement after the fact because of the order in which sub areas are generated but anyways.

Then if we can simply figure out which quadrants our drone is currently in we can potentially reduce what portion of our sub area list we search to just a quarter.

To do this we can actually just use our check_overlap function with our drone and a square that exactly encompasses each quadrant. Now typically because the quadrant is smaller than the square you wouldn't be allowed to say just because the drone is in the square means it's in the quadrant. But remember we've implemented our trajectories so that the drone never leaves the circular area so it works here. After getting which quadrants the drone is in we enter one of the ugliest pieces of code which, based on which combination of quadrants the drone is in, returns either one or two pairs of indexes corresponding to which portion of the sub area list should be searched. The reason we might need two pairs of coordinates is because let's say the drone is in quadrants one and four. Our sub area list does not wrap around so we must search the first and last quarter of the list splitting the portions of the list we must search. You must also note that diagonal combinations are impossible, a drone cannot cover just quadrants one and three or two and four. So these are not checked for in the if statement tree. And of course combinations are redundant so for example if you checked for quadrant combination one and two, you do not need to check for two and one.

To make this process easier to understand I created something akin to a state machine which I will include here. Note that I say akin to a state machine, it's important to note that the order in which we traverse the tree matters. We first check quadrant one and if true all of its possible combinations first. Then quadrant two minus any combinations with quadrant one, then quadrant three minus any combinations with one or two etc… Kind of like a depth first search.



Through this we can determine which portions of our sub area list to actually search. Now this does require extra effort, and I have found with a small number of sub areas the difference in performance is negligible. However it does not take that many sub areas to start noticing substantial improvements as the number of sub areas grows exponentially with linear changes in sub area size whereas deciding which quadrants to search is a flat cost. In my testing, moderate simulations saw about a 40% decrease time to calculate updates but this number as mentioned really depends on the number of sub areas. Now personally I imagine there are even better ways to reduce the sub areas we search but to me this felt like the most bang for buck approach.

check_overlap

This function simply takes a circle and a square and checks if they overlap. Now as much as I'd like to say I came up with this conceptually myself I really only came up with the literal corner case checking before I went searching online. This led me to a stack overflow question similar to mine in which user eJames details in his response a very efficient and straightforward method to achieve this detection. Please refer to eJame's explanation, stackoverflow.com/questions/401847, and note this is used under the CC BY-SA 3.0 license. Also note eJames only includes pseudocode and my code is my own implementation of the algorithm he describes in his response, not an adaptation of code.

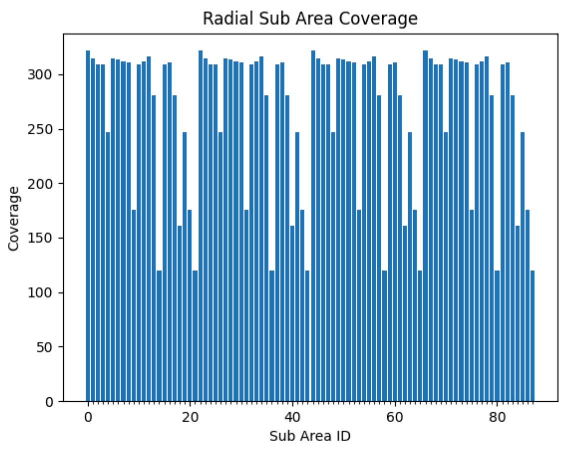
sub_area.py

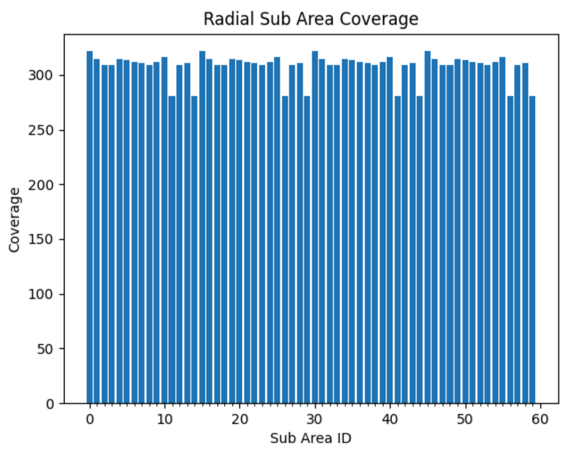
Ending off on a much simpler note the sub area file contains the sub area class with the only function being its constructor. Each sub area gets an ID corresponding to its index in the sub area list and a covered attribute to keep track of coverage received. Remember how we created four sub areas with flipped coordinates to tile every quadrant at once in our create_sub_areas function. Well here is where we flip the coordinates based on the quadrant number handed in. Now because our coordinates define the top left and bottom right of our sub area we calculate the center coordinates of the sub area ourselves for future reference. Then we create a corresponding square on our canvas.

Results

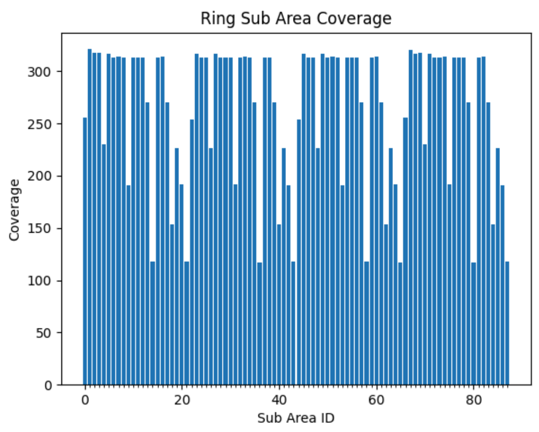
The results are pretty lackluster across all trajectories implemented, despite my best efforts with favorable parameters. However I do believe that the program does succeed conceptually however the failing comes in the way I do sub areas. The disconnect comes from the fact that the trajectories are based on a circular area but the checking coverage is based on the sub areas which do not perfectly tile the circle. This was obvious from the start and I suspected it would cause issues however looking at the coverage results it is more impactful than I'd hoped. The results shown are using a circular area radius of 100, drone coverage radius of 10 and side area length of 20. Because the trajectories are deterministic, repeat cycles simply scale results.

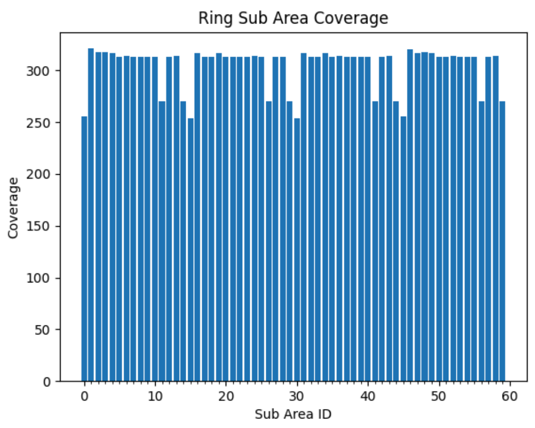
But to actually know if this is the problem I made the generate_sub_areas_less function which does not create sub areas that will be cut off. Although this does defeat the point of tiling a circular area these new results speak for themselves. Using the same parameters the coverage is much much more even which to me illustrates that in fact these cut off sub areas are the real problem but the trajectories themselves have been implemented fairly successfully.





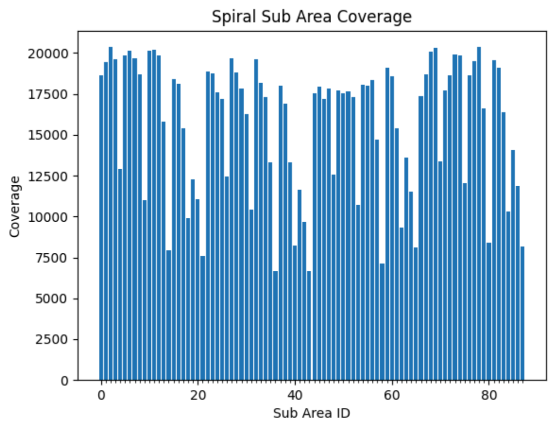
The results with the ring trajectory are very similar, again with the similar parameters besides trajectory the full tiling results in lackluster results, but using the generate_sub_areas_less function instead yields much more even coverage results.

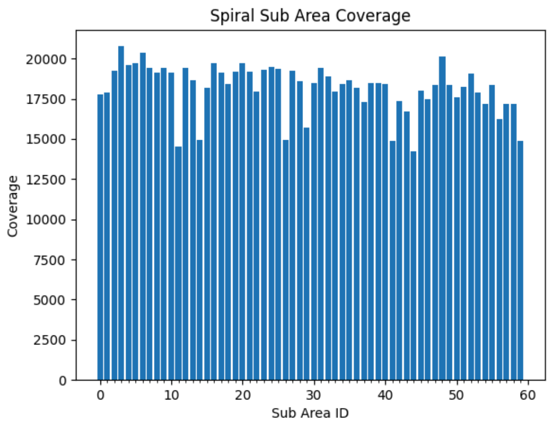




One thing to note is that the coverage values are symmetrical between quadrants with the deterministic radial and ring trajectories. This is seen in graphs as the coverage values repeat four times exactly corresponding to the sub area ids for each quadrant.

Lastly is the spiral trajectory which is a bit of a different situation. Whereas the radial and ring trajectories are deterministic the spiral trajectory uses random angles to rotate the trajectory thus more extensive testing is needed. In this case, increasing the number of cycles is the best way to ensure more accurate results, in this case I choose 10 cycles keeping the other parameters the same. Again we see that the tiling without cut off sub areas results in more even coverage but the results are more inconsistent.





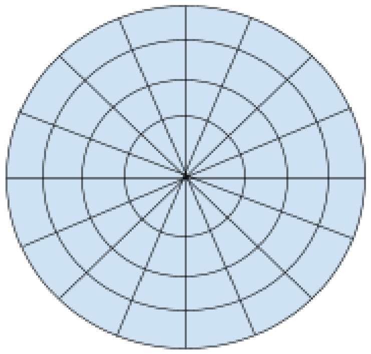
Lastly if you have chosen to run the simulation multiple times using the run again option within the program you may very well have noticed a considerable reduction in the window size. This is a rather strange bug that is only apparent when the matplotlib graphene section is included. I had a couple ideas about why this might be happening but nothing helped other than simply removing the matplotlib code. This is a very strange interaction between libraries, when the window size changes everything scales accordingly andt the results appear as normal.

Moving Forward

Hypothetically if someone wished to continue or further develop this program or its premise in some regard how might you begin. Well first off consider ditching the visualization and rewriting the entire simulation in C, this will offer a lot better performance which at the very least will make it easier to obtain more accurate results. As the accuracy of the simulation increases with the number of drones and the smaller sub area sizes. Frankly this is how I would probably do this program over, you could save your results to a file then run a bash script on it to feed it to a graphing tool or whatever to still get visualized results. Now this is not to say that Python or the visual aspect was a bad choice, both the choice of Python and creating simulation visuals helped immensely in getting the concepts programmed and debugging. Going straight into C with the same goals would have been extremely difficult but using the Python program to develop a C version would make the process much easier.

But in terms of actual features and changes to make, although technically language library dependent, there are a couple things you could do. You could try and fix the window size bug and figure out why the matplotlib and Tkinter libraries are messing with each other.
You could also add more trajectories as not all of the ones from the paper were implemented. But that's a big chunk to bite off so for some smaller stuff you can always mess around with the results graph and/or tkinter canvas to add more information or measurements.

You could also attempt a better way to tile the circle, perhaps slices split by rings as shown in the diagram. If this does result, which is likely will, in sub areas of different sizes just remember to scale the coverage each receives accordingly. Sadly this was not feasible as the actual area of overlap when a sub area gets cut off is not easily, maybe not even possibly, calculated unless certain symmetries are met.



Lastly there is something I actually had for the spiral trajectory at first but is applicable to all trajectories will take some extra effort. It is quite possible to calculate all the x and y coordinates each drone will go through as soon as the user has made their inputs. If you calculate these coordinate pairs before the visual simulation begins and store them in a list for each drone then all the update has to do is move through the right number of coordinates. Now this is complicated with changing velocity but is certainly possible and would mean the visual simulation could run a lot faster as most of the work has been frontloaded.

References

S. Enayati, H. Saeedi, H. Pishro-Nik and H. Yanikomeroglu, "Moving Aerial Base Station Networks: A Stochastic Geometry Analysis and Design Perspective," in *IEEE Transactions on Wireless Communications*, vol. 18, no. 6, pp. 2977-2988, June 2019, doi: 10.1109/TWC.2019.2907849. keywords: {Trajectory;Stochastic processes;Base stations;Wireless communication;Geometry;Throughput;Energy consumption;Aerial base station;trajectory processes;binary point process;stochastic geometry},

eJames, (2008, Dec 31). *Circle-Rectangle collision detection (intersection)*. Meta Stack Overflow. stackoverflow.com/questions/401847/circle-rectangle-collision-detection-intersection

https://www.desmos.com/calculator/lew2tky16w