Air Traffic Control System Simulation
Design Challenge
October 22, 2022
Project by Forrest Herman

Project Specifications Summary

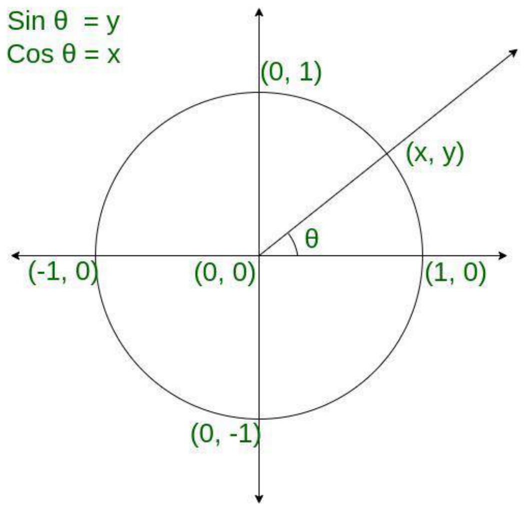
- The traffic control zone is a circle of radius 10km.
- The ATC must:
    - track of all airplanes' positions (latitude, longitude) within the traffic control zone.
    - queue airplanes for landing based on time of arrival into the traffic control zone
    - command airplanes to go for landing (when it is their turn) to one of two runways
        - otherwise, command airplanes to fly in a circular "holding pattern" which is a circle with a radius of 1km around a suitable point
    - never allow for planes to come within 100m of each other
- Airplanes have the following characteristics:
    - Assume each plane is a point.
    - When an airplane appears/spawns in the world, it does so at a random point on the edge of the traffic control zone, and it travels towards the center of the zone.
    - Airplanes travel at a speed of 140 m/s while flying. Assume that they instantaneously stop and disappear when they reach the end of the landing strip.
    - Airplanes automatically transmit their position (latitude, longitude) at a rate of 10 Hz to the ATC system while flying. They stop transmitting their position once landed.
    - When given a command from ATC to land at a specific runway, an airplane travels to the base of the runway in a straight line. Once it reaches the base, it turns/changes angle instantaneously to follow the path of the runway till the runway's end. Its speed is the same (140 m/s) throughout.

Brainstorm and Planning Phase

We can imagine the traffic control zone as a unit circle (this means angle 0/360 degrees is to the right (East), 90 degrees is upward (North), 180 degrees is to the left (West), etc. and the origin of the circle is its center (0,0). For each plane, we will need to store both a position and direction. This means each plane should have its own coordinates (x, y), and direction (θ) values. To achieve this, we will use the class structure in Python.

To spawn the planes in a circular fashion (at random) we will first get a random angle between 0 and 2π in radians. Then, using the zone radius and trig, we can determine the coordinates of the point:

$$x = r_{zone}\cos(\theta_{rand})$$
$$y = r_{zone} sin(\theta_{rand})$$



We want each plane to spawn facing the centre (0,0), so with some trig we can calculate their direction based on x and y, which are randomly generated.

$$\theta = \tan^{-1}\left(\frac{y}{x}\right)$$

Note: in Python, all trig functions assume angles are in radians, thus we will keep our values in radians and only convert for human readability (as needed). We will also be using the *atan2(y,x)* function in Python's math library as it preserves the angle based on our current quadrant (i.e., it takes negative coordinates into consideration).

Flying/Movement Calculations

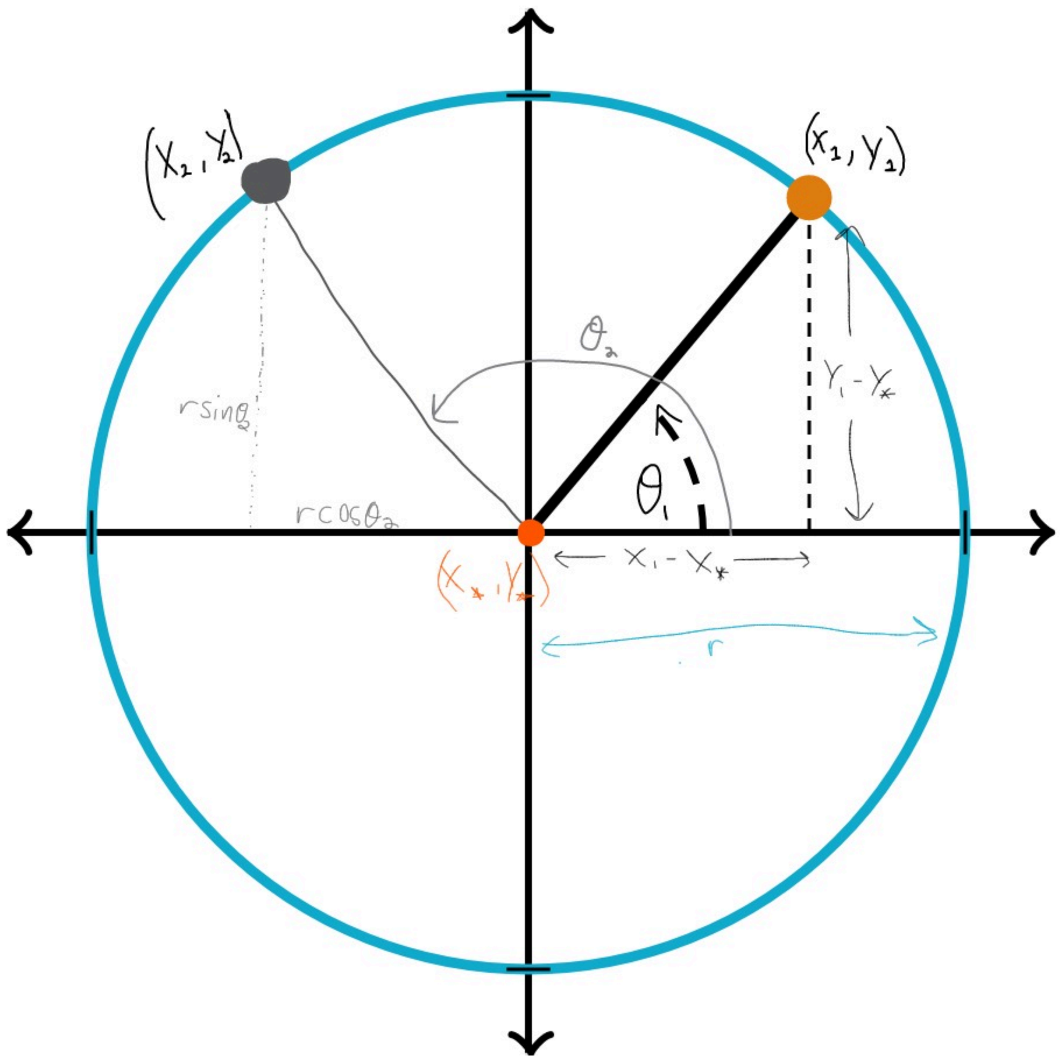
There will be 2 types of movement needed:

1. **Move directly forward towards a target point at the given speed (linear motion)**
   - This entails 2 options: either flying towards the airport or running down the runway.
   - Say our current position is (x_1, y_1) and our target point is (x_2, y_2). To accomplish this type of movement, we will need a movement vector, which we can calculate as $(\Delta x, \Delta y) = (x_2 - x_1, y_2 - y_1)$. Then, using the *atan2(y,x)* function once more, we can determine the new angle of approach (θ) for the plane.
   - We can then increment the plane's current position using the following formulas $x = x + v \times t \times \cos\theta$ and y$= y + v \times t \times sin\theta$, where v is the plane's speed and t is the time increment for the movement.

2. **Circle a point at a given radius (circular motion) ensuring the plane still moves at the given speed (account for angular velocity)**
   - This option was more complex. Assume the plane's current position is once more (x_1, y_1) and it will move to (x_2, y_2), which must be found. The approach I took was to lock onto a target point (with cartesian cords denoted (x_*, y_*) in the entire ATC zone but take it as my new reference point) and then find the next coordinate of the plane's motion by using the angular velocity, the circle's radius, and the time increment.
   - First, we need the angular velocity $\omega = \frac{v}{r}$, the plane's linear speed divided by the circling radius. Using this we can calculate the next angle relative to our target point: $\Delta\theta = \omega t$.
   - If we add that to our original angle relative to the target point, $\theta_1 = atan2(x_1 - x_*, y_1 - y_*)$ we can find $\theta_2 = \theta_1 + \Delta\theta$.

- To find the new position of the plane in the entire ATC zone, we use our new angle with trig rules and simply add the relative position of the target point:

$$x_2 = r\cos\theta_2 + x_*$$
$$y_2 = r\sin\theta_2 + y_*$$



Pathfinding and Logic

To determine when to perform certain actions, (when to circle, when to land, etc.) we need logic. The parameters for this I decided are as follows:

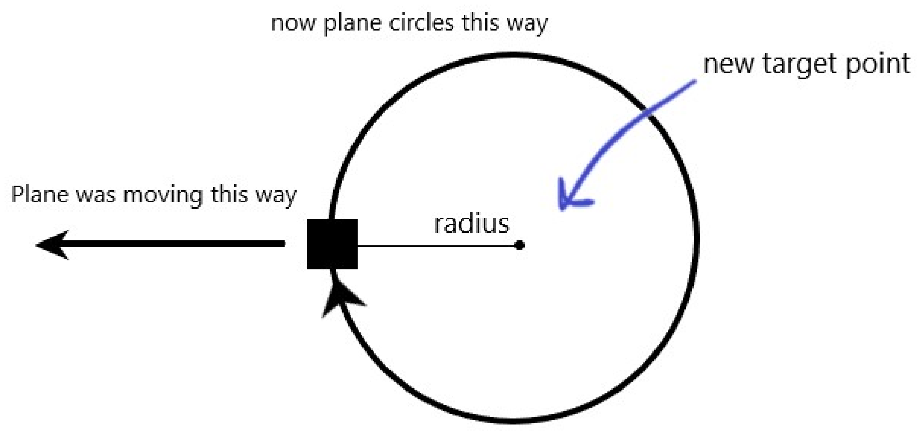
- Planes should get as close to the runway as they can before starting their circular motion.
- The circling motion should trigger if:
    1. The plane gets within range of another circling plane
    2. The plane gets within range of a runway

This means that planes with higher priority on the queue (earlier arrival time) will be closer to the runway when it becomes available. It also means planes won't run into each other unless the spawn too close together from the get-go.

- This logic is performed in the *plane.close_to_runway_or_hold_point()* method.

When a plan is within a tolerance amount to it's target runway, it will turn and roll down the runway. When it passes a certain distance on the runway, it is removed and counted as "landed". This is check is done in the *if plane.status == RUNWAY* conditional in the main while loop.

On the other hand, if it is determined that the plane should execute the circling behaviour, the *plane.find_hold_point()* method comes into play. It uses its current path to track backwards the amount of the circling radius and begin it's circular movement. This is shown in the following diagram:



Code File Structure

For this project, the file structure has been established as follows:

- *main.py*: the main loop running the simulation. This performs all the conditional checks using the parameters found in the other modules.
- *models.py*: a place to store the classes used, such as ATC, Plane, Runway, etc.
  - **class ATC**: this class holds all the main options and values, and is responsible for spawning planes, runways and other miscellaneous tasks. It also finds which runways are free, and tracks the planes positions as they move.
  - **class Runway**: this holds the positions of each runway, and fetches the coordinates for either the North or South side. It also stores whether it is available or busy.
  - **class Plane**: this class is responsible for the plane's movement, based on its current position, angle and/or target point (which it stores). While these target values are sent by the ATC, each plane calculates its own future position using these values. The class also tracks the status of the plane, be it FLYING, HOLDING, LANDING, or on the RUNWAY.
- *statuses.py*: stores helpful variables that are used to track the status of planes and runways and makes the code readable.
  - Plane statuses:
    - FLYING means the plane is yet unassigned to any task. It is simply flying towards the origin.
    - HOLDING means the plane is executing its circling maneuver.
    - LANDING means the plane has been assigned a runway and it on its way.
    - RUNWAY means the plane is currently on the runway and is about to be landed.
  - ATC statuses:
    - AVAILABLE means there is at least 1 open runway.
    - BUSY means all runways are taken.
- *simulation.py*: stores the Pygame contents and functions used to visualize the simulation. There are sprites and functions to update their positions as well as build the traffic control zone and scale the coordinates according to the screen size.

Debugging and Visualization

In order to debug my planes' pathfinding, I knew I wanted a visual interface. I briefly considered Tkinter, but it's more designed for building GUIs, so instead I turned to Pygame. Pygame allowed for regular refreshes of the planes' positions, redrawing them every frame at new coordinates. For my use case, that's all I needed.

Account for the following considerations when designing the system

1) ***An arbitrary number of airplanes should be able to be spawned (up to the maximum limit for the traffic control zone area, of course)***
   - By pressing space, I can spawn any number of planes. If planes are unable to spawn due to space constraints, we know we have met the zone limit. Alternatively, one can set the maximum when the zone is created to limit the number of planes further.

2) ***How would you allow for any of the above parameters to change without changing the code? (e.g. change number of runways, runway lengths or positions, or traffic control zone).***
   - All parameters are easily modifiable. I designed with modularity as a priority. The runways can be resized, scaled, etc. and the ATC will compute the coordinates accordingly. I do however always assume the runways are to be placed centered on the origin.
   - If the number of runways increases, we would want to fill the ones closest to the center first and work our way outward. This is done using the *prep_for_landing()* function in the *modules.py* file. However, the logic isn't perfect. If the only runway available is the farthest one, the plane will be forced to cross over the other runways.
   - The traffic control zone can be scaled up or down, that is built into the code. This is done by passing the parameters through the classes and using relative positioning.

3) ***What is the best way to transmit messages between the airplanes and the ATC?***
   - The real-world options that come to mind are high frequency radio channels or satellite communications. This allows the plane to communicate over the 10+km distance, and give their updates with a short amount of latency. Radio can work well since the planes are in the sky with minimal interference and often a clear line of sight with ATC towers and antennas. Satellite control of course is more versatile and works with access to the sky to transmit messages worldwide.
   - If you meant in terms of the code structure, then I think my current implementation of communicating through the ATC class makes sense. The ATC holds all the parameters and keeps track of which runways are free and takes input from the planes while also giving them direction. This follows a hub architecture (all devices communicate through a common device) which simplifies each plane's computation by offloading it to the ATC, and each plane only must communicate to one place, which simplifies their job.

4) ***What is the best way for the ATC to set holding patterns and paths for the planes such that they will not come too close to each other?***
   - The method I found was to determine if either
     1. a runway was free, or

ii. if there was already a plane in the vicinity who is performing a holding pattern.

Using this method, we are looking for a proximity to either an occupied runway or an existing plane in a holding pattern which will trigger the current plane to execute a holding pattern.

- Some issues with this method are if 2 planes spawn too close together. Since they are already within range, they might both execute holding patterns that are not adequately spaced apart. This may or may not be an issue, but personally, based on real-world examples, I don't expect planes to come into view side by side.
- The other issue faced with this method is if too many planes are waiting, and many planes come from the same direction, that side of the zone may become cluttered. As such, this method of mine assumes the planes will be evenly spaced out, coming from all directions. (Which in fact is unlikely in practice, but also even just 10 seconds between planes should resolve any issues, so it's not so impractical).

Project Demos

Demo 1 (2 runways)

Demo 2 (4 runways)