3D Random Walk

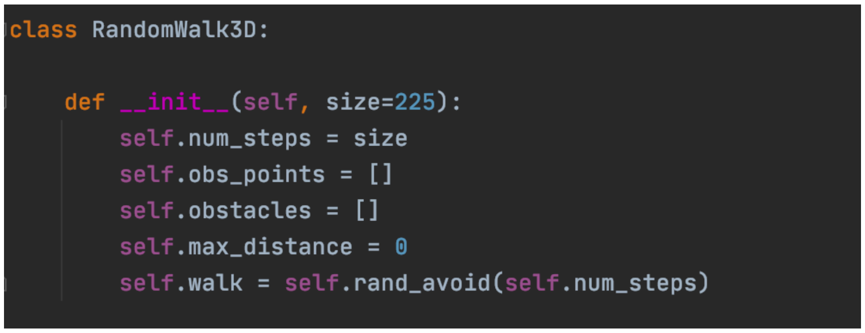
Introduction

In this project, I created a program that can create a 3D random walk that avoids the borders, random obstacles, and itself. Using it is able to detect and avoid dead ends. Using Recursion it is able to avoid any route that will lead to a dead end before the full amount of steps have been taken.

Implementation

In order to make everything work correctly, I first began with a 2D version of this program that works under the same premise. In fact the main method that does the recursion is exactly the same for the 2D and the 3D version of the program.

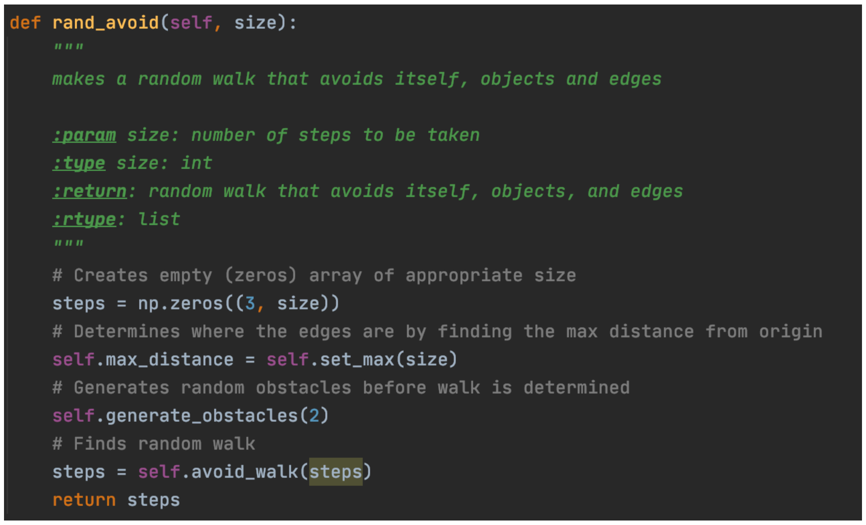
To start off, I make an object RandomWalk. This way I can give it certain attributes that make the computation time shorter than if I made this a pure script. Certain attributes like number of steps or the border that surrounds the walk are helpful for multiple functions. In its constructor I go ahead and make it run the walk to get the steps. All that is left is to call the show walk function to see how the walk did.



```python
class RandomWalk3D:

    def __init__(self, size=225):
        self.num_steps = size
        self.obs_points = []
        self.obstacles = []
        self.max_distance = 0
        self.walk = self.rand_avoid(self.num_steps)
```

When it makes the random walk it starts a function that essentially acts as a managing function. It creates the steps array, creates the border, and makes the random obstacles that the walk will have to avoid. This function simply takes the number of steps that are needed and will return a finished walk ready to be displayed later.

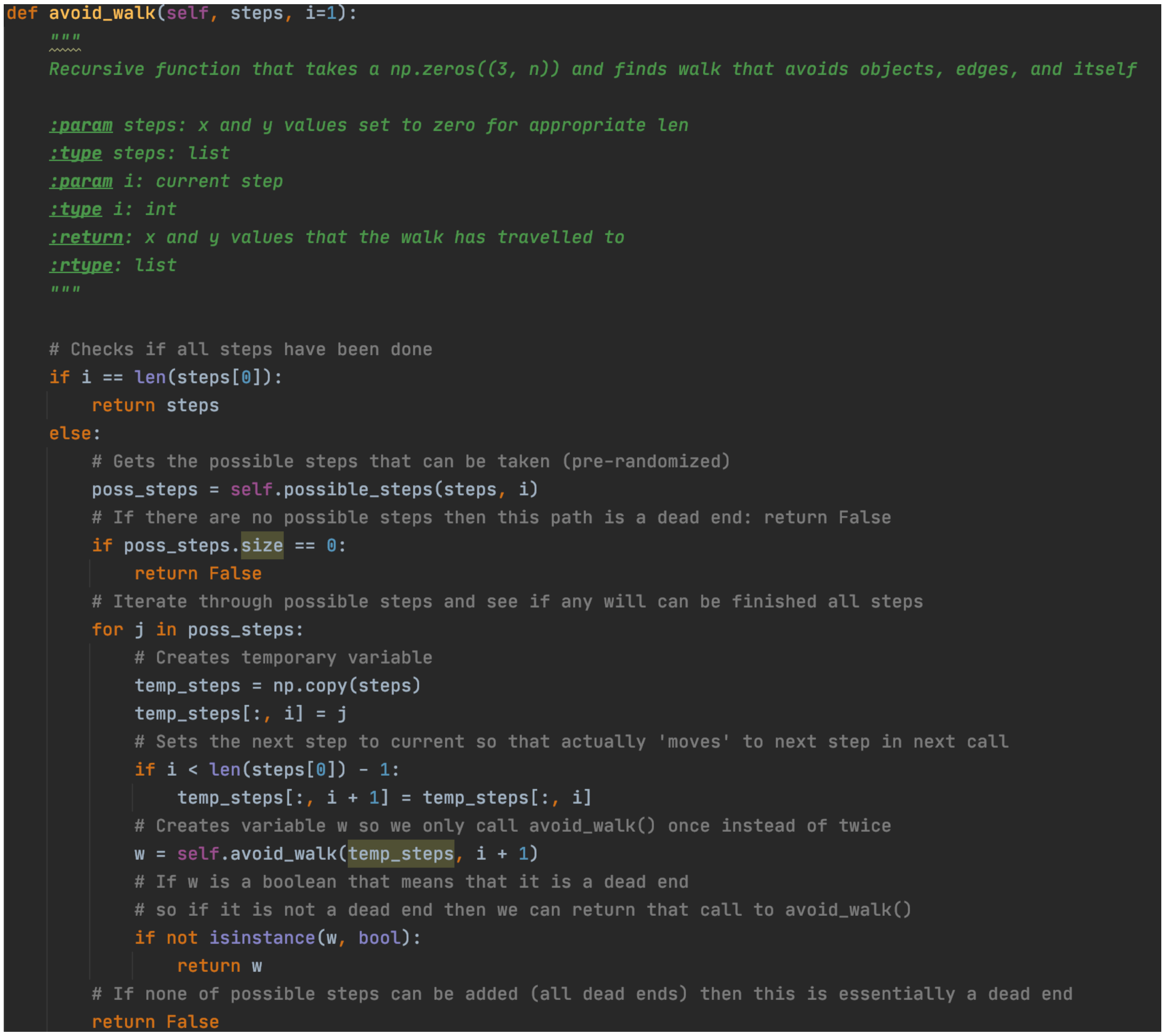


```python
def rand_avoid(self, size):
    """
    makes a random walk that avoids itself, objects and edges

    :param size: number of steps to be taken
    :type size: int
    :return: random walk that avoids itself, objects, and edges
    :rtype: list
    """
    # Creates empty (zeros) array of appropriate size
    steps = np.zeros((3, size))
    # Determines where the edges are by finding the max distance from origin
    self.max_distance = self.set_max(size)
    # Generates random obstacles before walk is determined
    self.generate_obstacles(2)
    # Finds random walk
    steps = self.avoid_walk(steps)
    return steps
```

This function directly calls the main part of this program that is the recursive 'avoid_walk'. This function implements a recursive algorithm in order to find a random walk and returns the final result. This function follows the following structure

1. Check to see if we have reached the limit for the amount of steps to take. If so, simply return the finished steps.
2. Now fetch the current possible steps. If there are none, then return False to indicate that we have reached a dead end and cannot go on.
3. For each of the possible steps (the possible steps function returns the steps in a random order so that the recursive function doesn't have to deal with it) check to see if we can solve the rest of our walk by calling 'avoid_walk' with the updated steps taken. If so, then return the call to 'avoid_walk' (as it will have the updated steps).
4. Lastly it returns False. This is so that we can avoid going ways that would ultimately result in a dead end before the program reaches the ends of the steps.

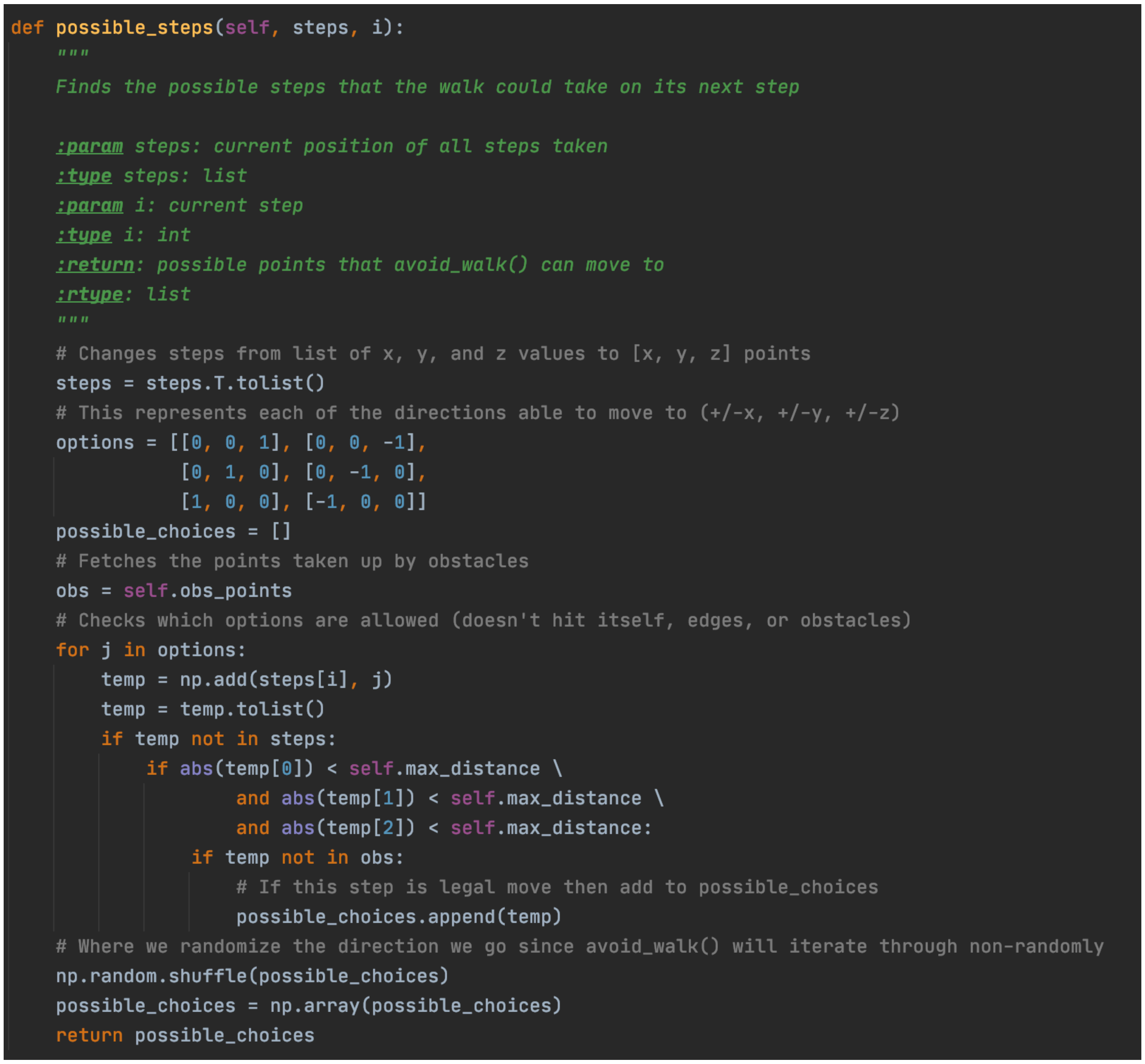


```python
def avoid_walk(self, steps, i=1):
    """
    Recursive function that takes a np.zeros((3, n)) and finds walk that avoids objects, edges, and itself

    :param steps: x and y values set to zero for appropriate len
    :type steps: list
    :param i: current step
    :type i: int
    :return: x and y values that the walk has travelled to
    :rtype: list
    """

    # Checks if all steps have been done
    if i == len(steps[0]):
        return steps
    else:
        # Gets the possible steps that can be taken (pre-randomized)
        poss_steps = self.possible_steps(steps, i)
        # If there are no possible steps then this path is a dead end: return False
        if poss_steps.size == 0:
            return False
        # Iterate through possible steps and see if any will can be finished all steps
        for j in poss_steps:
            # Creates temporary variable
            temp_steps = np.copy(steps)
            temp_steps[:, i] = j
            # Sets the next step to current so that actually 'moves' to next step in next call
            if i < len(steps[0]) - 1:
                temp_steps[:, i + 1] = temp_steps[:, i]
            # Creates variable w so we only call avoid_walk() once instead of twice
            w = self.avoid_walk(temp_steps, i + 1)
            # If w is a boolean that means that it is a dead end
            # so if it is not a dead end then we can return that call to avoid_walk()
            if not isinstance(w, bool):
                return w
        # If none of possible steps can be added (all dead ends) then this is essentially a dead end
        return False
```

Now to show how we are determining the possible steps. Simply put, we are applying each of the ways that the walk could go from the current step (up, down, forward, backward, left, and right). But we have to make sure that the walk doesn't collide with itself, the edges, or any obstacles that might be there. So what we do is have it check to see if applying that change would put it in one of the points taken up by itself, the edge, or an obstacle, and if it is not taken up by one of those things then we can add it to a list of points that don't hit anything either. This list is then shuffled and returned to be used by the 'avoid_walk'.

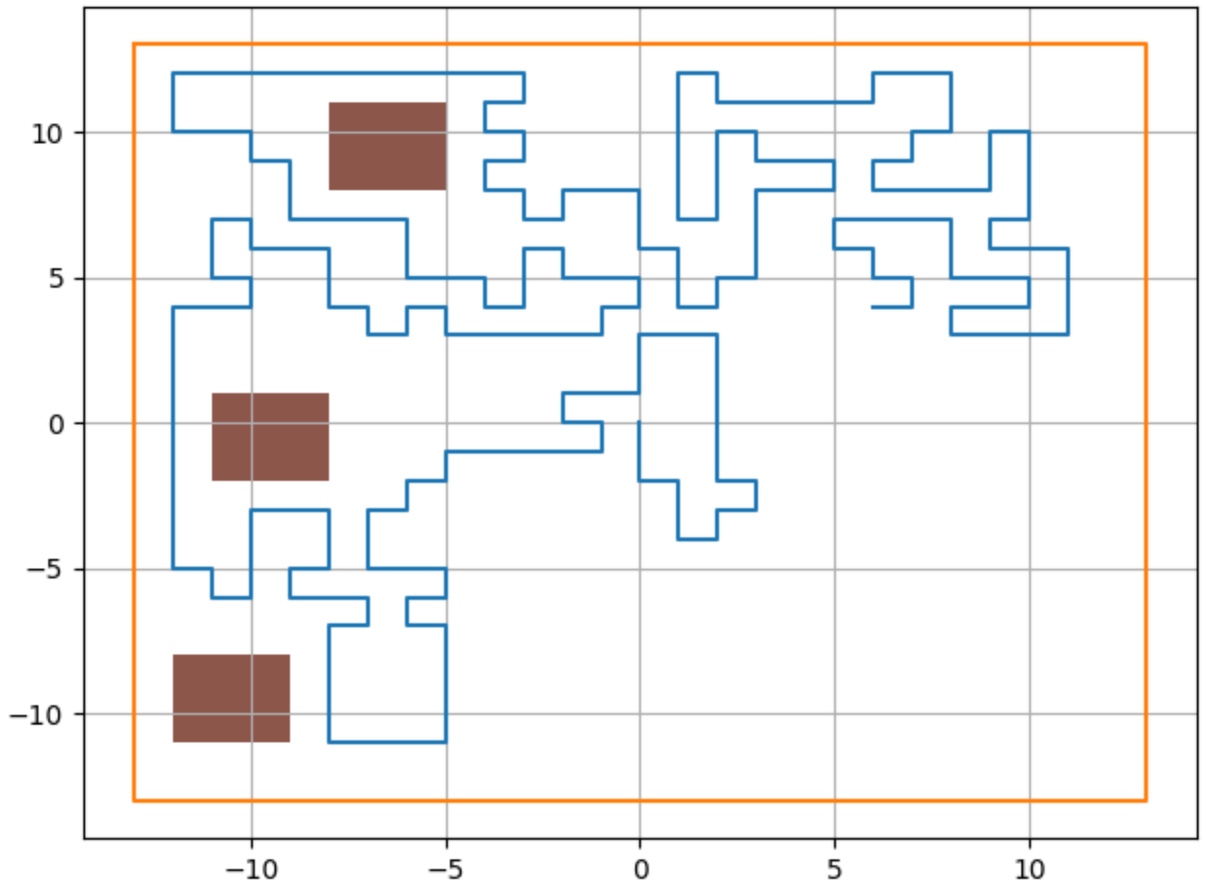


```python
def possible_steps(self, steps, i):
    """
    Finds the possible steps that the walk could take on its next step

    :param steps: current position of all steps taken
    :type steps: list
    :param i: current step
    :type i: int
    :return: possible points that avoid_walk() can move to
    :rtype: list
    """
    # Changes steps from list of x, y, and z values to [x, y, z] points
    steps = steps.T.tolist()
    # This represents each of the directions able to move to (+/-x, +/-y, +/-z)
    options = [[0, 0, 1], [0, 0, -1],
               [0, 1, 0], [0, -1, 0],
               [1, 0, 0], [-1, 0, 0]]
    possible_choices = []
    # Fetches the points taken up by obstacles
    obs = self.obs_points
    # Checks which options are allowed (doesn't hit itself, edges, or obstacles)
    for j in options:
        temp = np.add(steps[i], j)
        temp = temp.tolist()
        if temp not in steps:
            if abs(temp[0]) < self.max_distance \
                    and abs(temp[1]) < self.max_distance \
                    and abs(temp[2]) < self.max_distance:
                if temp not in obs:
                    # If this step is legal move then add to possible_choices
                    possible_choices.append(temp)
    # Where we randomize the direction we go since avoid_walk() will iterate through non-randomly
    np.random.shuffle(possible_choices)
    possible_choices = np.array(possible_choices)
    return possible_choices
```

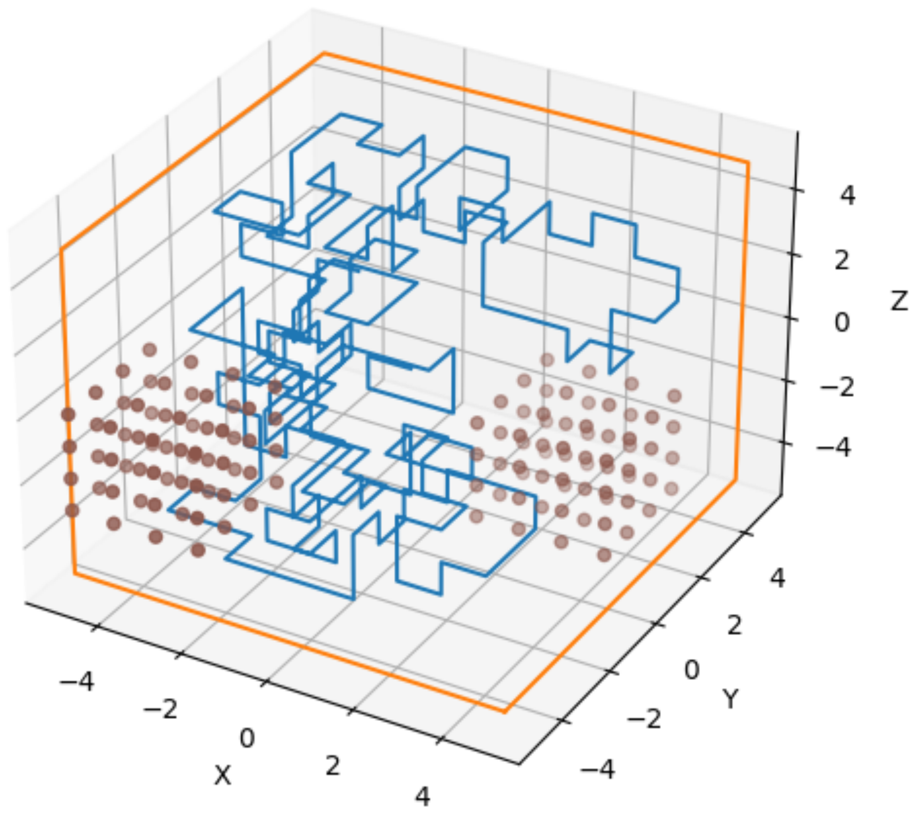
Results

When this is run and plotted it is very difficult to see where the line is going and difficult to determine whether or not the line is actually avoiding stuff because of the confusion. Here is a 2D version of the results that we can inspect first.



Here is an example of the walk that is easier to look at. First, we can see that the walk is enclosed within the yellow border. This was necessary to make sure that the objects would placed within a reasonable distance of the line. It also adds an interesting complication to the task of finding a walk that doesn't result in a dead end. We can see that it starts at the origin and goes whichever way. But when it gets close to the objects the choices are narrowed and it travels in a much more predictable way. The important thing from this example is that it is easy to inspect and shows that the proof of concept is working.

Next we need to try the 3D walk. This should be easier in some ways because it won't have as many dead ends, and for the 2D version this sometimes caused complication because it could take too long to solve because of the amount of options that would end in an eventual dead end.



The 3D walk as you can see is much more confusing to look at because it is difficult to tell the depth of the lines. For this reason I made the obstacles be shown as dots instead of solid boxes. This way we can see that if the line were to intersect with one of the obstacles it would have to pass through one of these points. The border is also only shown along the planes in order to make the walk more visible.