👍👍👍👍👍👍👍👍🤖👍👍👍👍👍👍💩👎💩👍👎💩👍👍👍💩💩💩👍👎💩👍👍👍💩💩👍👍👍👍👍👍💩

Disclaimer

I genuinely just wanted to have fun with this assignment so it is definitely not at all professional. Nonetheless, I made an effort to address all the requirements amidst the fun. As such, I totally understand any penalties associated with professionalism.

👍👍👍👍👍👍👍👍👍👍👍👍👍👍👍💩👎💩👍👎💩👍👍👍💩💩💩👍👎💩👍🤖👍👍💩💩👍👍👍👍👍👍💩

Introduction

The zoomba bot is every college student's best friend, if they can afford it. The tiny little mash of metals, plastic and physics does our cleaning for us automatically. Bless the folk at iRobot for this. The device is created to be a general service robot so it is not tailored to any house layout. You simply purchase one, put it on your floor and it starts to move around cleaning the floor.

Sadly, there is no sure way strategy or pattern that allows it to cover any layout give. This is because, in Its base state, it has no memory or perception of the house. All it has access to is the information that Its sensors, which typically can only see the immediate vicinity of the bot, can perceive. This however, does not mean that we cannot try. iRobot patented a set of patterns that the first generation Roomba bots follow. These are the Random Bounce and Wall tracking algorithm. These two are the bases if this paper.

Sadly , I am one of those college students who cannot afford a Zoomba so, let us simulate one 🙂.

Setup Specifications

The system we design is simulated as follows. We have a grid which is our house. The dimensions of this are user specified. Atop this, we also specify the number of obstacles which the system will put in the house for the bot to navigate.We also need to specify how messy the house is. By default, the system randomly messes up the house.

The house generation is as follows. The system will generate a grid on the dimensions specified which is an m x n array where m is the number of rows and n is the number of columns. These will be initiated with a value between **0, 1,2 or 'W' for cleanliness** depending on the specification. Each grid has a class instance of tile which has this information. The tile class also saves information of Its **Von Neumann neighborhood.** At any given tile, the bot can move to any of these tiles and as such, the possible directions of motion are always **"Top", "Bottom", "Left" or "Right".**

More on the cleanliness, 0 is a clean Tile, 1 is a slightly dirty tile, 2 is a very dirty tile. Other information saved in the tile are the number of times it can be visited which is information that will be used with a more advanced roomba system. This is meant to emulate Roombas with WiFi connectivity. This connection enabled them to know if they have been in some section of the room and how many times they were there. A high reading indicates that the place was visited quite often and there is a lower likelihood of finding anything new and thus, our system favors exploring new cells. Ato this, we use the data to create a heat map.

For visualization, we have an inbuilt processor which takes in the grid and outputs the string representation of the house. What this does is map the cleanliness to a character in the map key provided. A very clean tile gets a 👍
, a **Slightly Clean** one gets a 👎, a **Dirty** one gets a 💩 and a **Wall/Obstacle** gets a 💀 to signal the danger associated with hitting a wall.

The user can specify the number of walls generated but to keep things sensible, we limit the number such that the maximum you can select is half the horizontal length, specified in the dimensions. This ensures that there will always be some space for non wall tiles. We also limit the length of the walls that can be generated so as to ensure that there is no wall combination so long it will close in the bot to one section of the room thus, there will always be space to move around. Other than that, these values are randomly generated with these constraints enforced.

Setup Specifications

Extensive comments are included in the code detailing how the bot works. Nonetheless, the interface is what matters and so here I provide a documentation of the different settings. In the collab, various configurations are tested.

When initiating a **house_c**, these are the parameters you work with.

Parameter	Value Type	Function
obstacles	Int **Default 5**	Specifies the number of horizontal obstacles. Constrained to be less than half the column specified in dimensions.
mess	String: [**"Random"**, "Very Dirty", "Slightly Clean", "Clean"] **Default "Very Dirty",**	Specify how messy you want the room to be.
dimensions	Tuple (int, int) **Default (15, 15)**	A tuple specifying the dimensions of our house grid.

Example initialization:

```
house1 = house_c(obstacles = 40, mess = "Random", dimensions  =(100, 100))
#Methods in the class
house1.spawn_house() #No extra parameters
```

When initiating a **roomba bot** , these are the parameters you work with.

Parameter	Value Type	Function
wall_walking	Boolean (True, False) **Default: True**	Set to true if you want the wall walking strategy applied
random_bounce	Boolean (True, False) **Default: True**	Set to true if you want the random bounce applied.
minimum_visits	Boolean (True, False) Default: True	Set to true if you want the minimum visits strategy applied.
start	Tuple (int, int) **Default (0,0)**	The starting position of the bot
house	**<class type house_c>**	Parse in an instance of the house_c class
delay	Int **Default 1**	Delay when blinking before clearing output and printing the next state

Example initialization

```
roomba1 = roomba(house = house1, delay = 1, start = (2,7),wall_walking =True, random_bounce =
True, minimum_visits = True)
roomba1.clean(clean_level=0, focus =True, blink =True, heatmap = True, out_steps=1)

#Methods in the class
roomba1.check_clean() #Returns total count of house dirtiness

#Takes in bot position.
#Returns a dictionary of the possible directions to move depending on strategies specified.
roomba1.directions(bot_position)

roomba1.heatmap() #Returns a plot of the bot heatmap so far in the house

#Initiates the bot movement and cleaning process.
roomba1.initiate_traversal(focus, clean_level, blink=True, heatmap=True, out_steps=20)

#roomba1.clean(clean_level, focus, blink, heatmap, out_steps)
```

Method parameters

Parameter	Value Type	Function
blink	Boolean (True, False) **Default: True**	This animates and delays. It blinks the house and heatmap to produce an animated effect of the bot in action.
clean_level	float <0, 1>, **Default 0.3(30% clean)**	How clean you want you house to be. If you want just 30% of the original dust level, you set it to 0.3. The lower the value, the cleaner the house will be.
focus	Boolean (True, False) **Default: True**	Tells the bot to stay in a tile until it is fully clean before moving on.
out_steps	Int Default = 1	The steps the bot makes before printing. For a smooth blink effect, set to 1. Increase for a faster simulation. Set to a very huge number if you just want the final state and statistics for faster(Data oriented) simulations.
heatmap	Boolean (True, False) **Default: True**	Produces a heatmap of the boot. It is nice when trying to understand why the bot almost always terminates and is also a superb visualization of the coverage.

Demonstrations

To minimize output clutter, I recorded videos with several configurations with the blink functionalities. I also discuss these and why the behaviour is as observed. The demonstrations are on a small house to keep them short.

For analysis, we will produce minimum output to keep the simulation fast and focus more on the performance of the different strategies in lieu of the behaviour of the strategies.
For the coverage heatmaps, I used a 50*50 simulation of a house with 20 obstacles.. This was to keep things fast. The simulation is in my notebook.

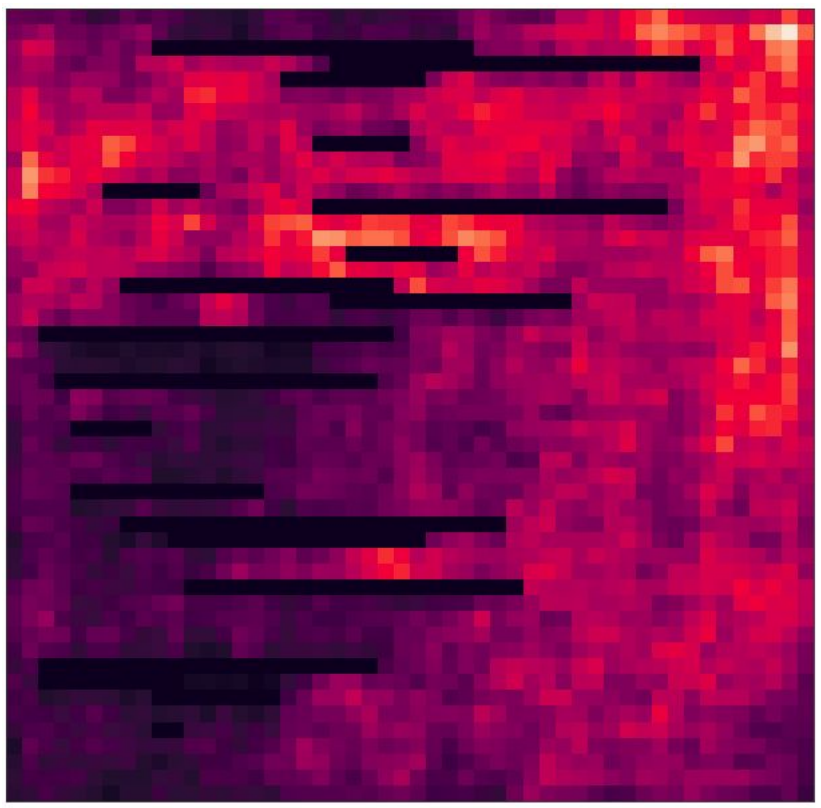
Random Bounce only, no wall walks or minimized visits.

* Configuration

```
house1 = house_c(obstacles = 5, mess = "Random", dimensions  =(10, 10))
house1.spawn_house()


roomba1 = roomba(house = house1, delay = 1, start = (0,0),wall_walking =False,
                 random_bounce = True, minimum_visits = False)
roomba1.clean(clean_level=0, focus =True, blink =True, heatmap = True, out_steps=1)
```

Video: https://www.youtube.com/watch?v=l2r1Fpfo-sg&feature=youtu.be

This is the least efficient combination of the approaches taken. The bot tends to get stuck in one zone.



* How it works

The algorithm essentially moves in a random direction until it hits a wall or encounters a dirty patch in which case it cleans then switches direction. This alone makes the bot seen like it is moving almost randomly especially when in a patch that is already clean.
Coverage was biased to open sections of the room as the system had no idea of where it had been so the bot would get to

a door, not know how to leave and went back in. This was admittedly a frustrating bot to cheer on.

Random Bounce + Wall Walk, no minimized visits.

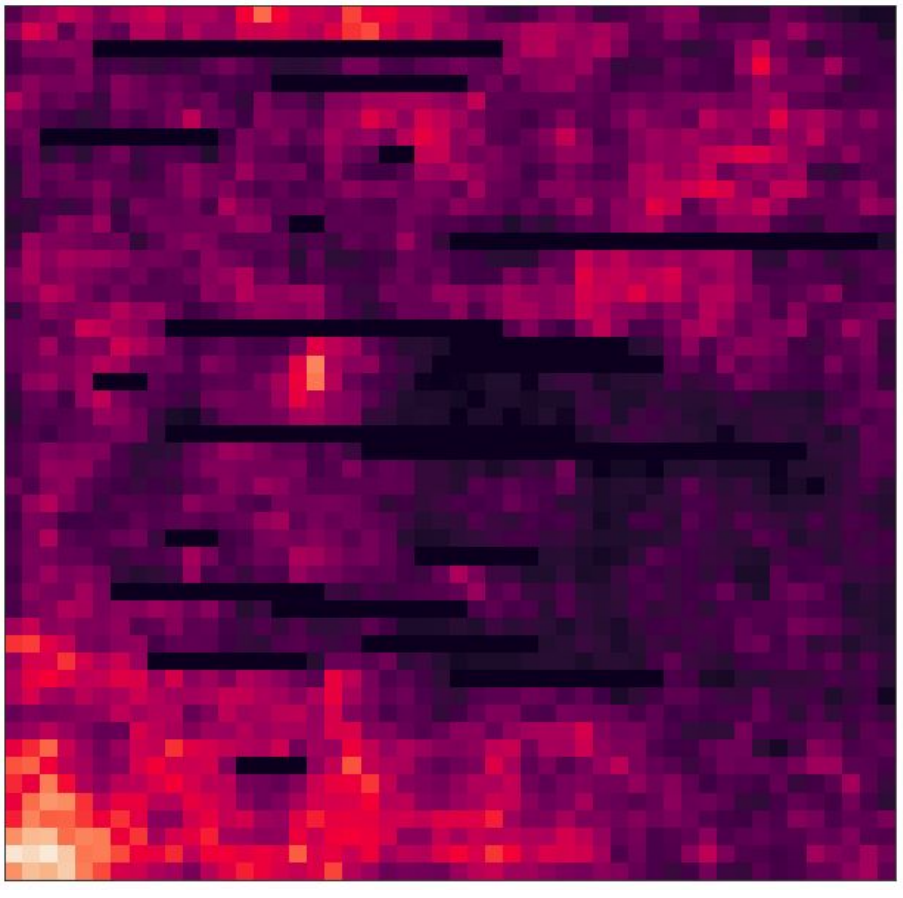
* Configuration

```
house1 = house_c(obstacles = 5, mess = "Random", dimensions  =(10, 10))
house1.spawn_house()


roomba1 = roomba(house = house1, delay = 1, start = (0,0),wall_walking =True,
                 random_bounce = True, minimum_visits = False)
roomba1.clean(clean_level=0, focus =True, blink =True, heatmap = True, out_steps=1)
```

Video: https://www.youtube.com/watch?v=4YYamNrZnJs&feature=youtu.be

* How it works

This was a more efficient combination of instructions. The bot favors walking along a wall but will jump randomly to a dirty tile if it encounters it. This was faster than the Random Bounce only.
Coverage also tended to be biased to regions confined by walls. You can see this in the heatmap to the right. This of course is because our bot does not know where it has not been.
Atop this, this was always about half as fast at the Random Bounce alone.
But now let us talk about our MVP.



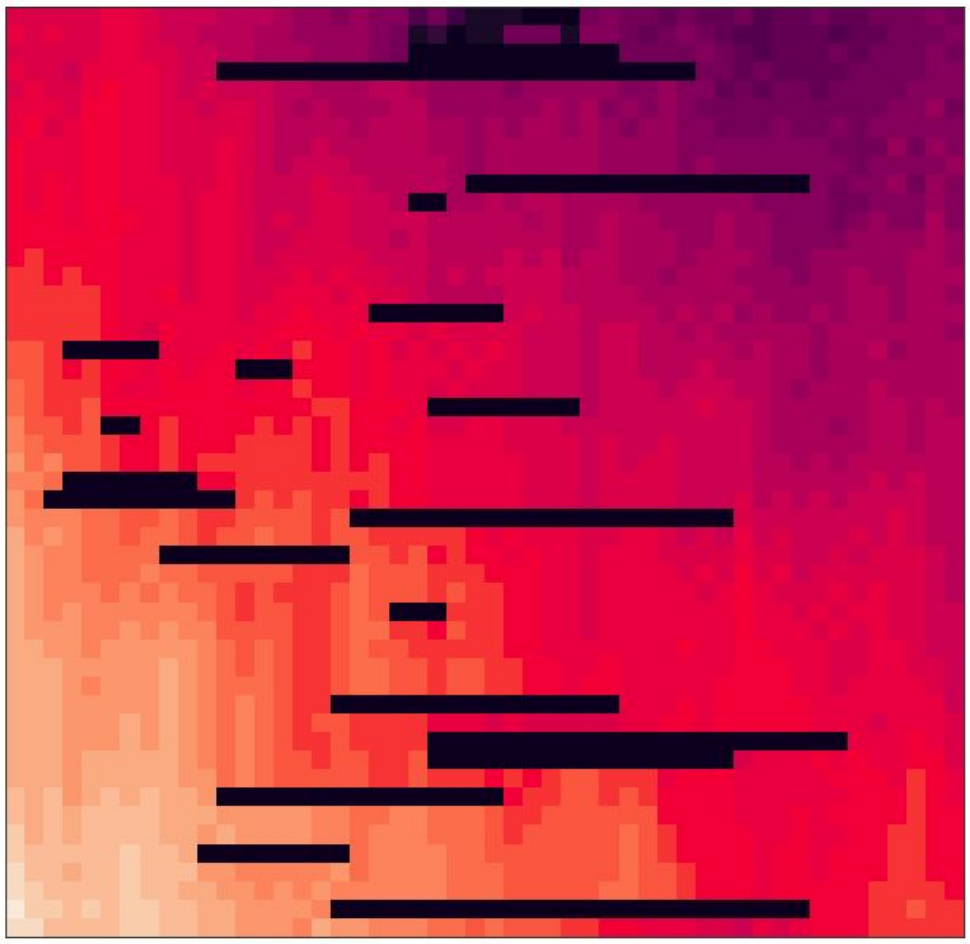
Random Bounce + Wall Walk, no minimized visits.

Video: https://www.youtube.com/watch?v=rRPVwPq5hn8&feature=youtu.be

*Configuration

```
house1 = house_c(obstacles = 5, mess = "Random", dimensions  =(10, 10))
house1.spawn_house()


roomba1 = roomba(house = house1, delay = 1, start = (0,0),wall_walking =True,
                 random_bounce = True, minimum_visits = True)
roomba1.clean(clean_level=0, focus =True, blink =True, heatmap = True, out_steps=1)
```

* How it works



This one had all the advantages the first two strategies offer but also it is able to tell how many times it has been in certain places. It thus requires a roomba that has some aspect of memory and this is provided by roomba's with WIFi connectivity. This system is efficient and very satisfying to observe since it does not waste a lot of time in places it is stuck in. It will always move to a neighbor it has never visited before or has visited the least.
Coverage is more spread in a somewhat gaussian manner with a biased to some region. This is the region where it got stuck and ended up having to escape. The gaussian is relatively flat though thus the spread scoped most of the room quite well. Note, the heatmap uses visit counts that are way lower than those for the other strategies so it is more sensitive., it cleared out the room in a quarter the time the others did.

Performance Test: Steps to clear the room

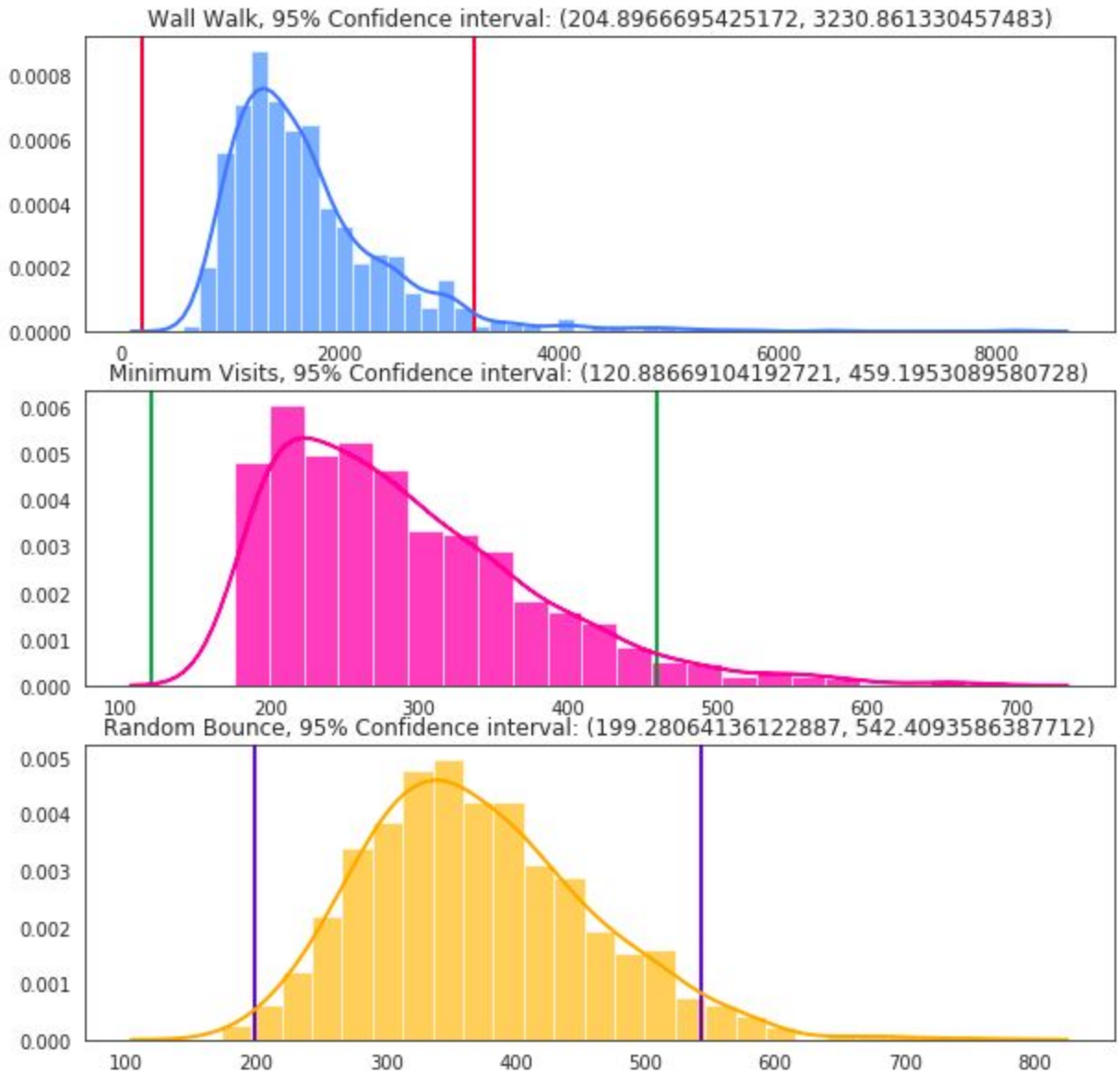
We have seen that the combined effect of the three strategies is generally better. We however, are also interested in the performance of the individual ones. For this, we will run tests as follows:
Run the algorithm 10000 times on a 10*10 grid and see how long it takes to clear 100% of the house(0 cleanliness). I used onlly 1 obstacle. Using more than 1 runs the risk that the obstacles, which are walls in our case, will be on two back to back rows and join in the middle to create a full wall which locks the bot in and this leads to an infinite loop. It took me 3 hours to figure that out 😭.

The house is randomly messed up at each instance and is placed with 5 obstacles.
This is the set up code:

```
random_bounce = []
wall_walk = []
minimum_visit = []

for i in range(1000):
    #Each house will have Its own instance since this is modified during cleaning.
    copy1 =  house_c(obstacles = 1, mess = "Random", dimensions  =(10, 10))
    copy2 =  house_c(obstacles = 1, mess = "Random", dimensions  =(10, 10))
    copy3 =  house_c(obstacles = 1, mess = "Random", dimensions  =(10, 10))

    #Spawn a house
    copy1.spawn_house()
    copy2.spawn_house()
    copy3.spawn_house()

    #Initiate the roombas
    roombaRB = roomba(house = copy1, delay = 1, start = (0,0),wall_walking =False,
                random_bounce = True, minimum_visits = True)
    roombaWW = roomba(house = copy2, delay = 1, start = (0,0),wall_walking =True,
                random_bounce = False, minimum_visits = False)
    roombaMV = roomba(house = copy3, delay = 1, start = (0,0),wall_walking =False,
                random_bounce = False, minimum_visits = True)

    #And lets dance!!!
    rb = roombaRB.clean(clean_level=0, focus =True, blink =False, heatmap = False,
out_steps=10000000, fin=False)
    ww = roombaWW.clean(clean_level=0, focus =True, blink =False, heatmap = False,
out_steps=10000000, fin=False)
    mv = roombaMV.clean(clean_level=0, focus =True, blink =False, heatmap = False,
out_steps=10000000, fin=False)
```

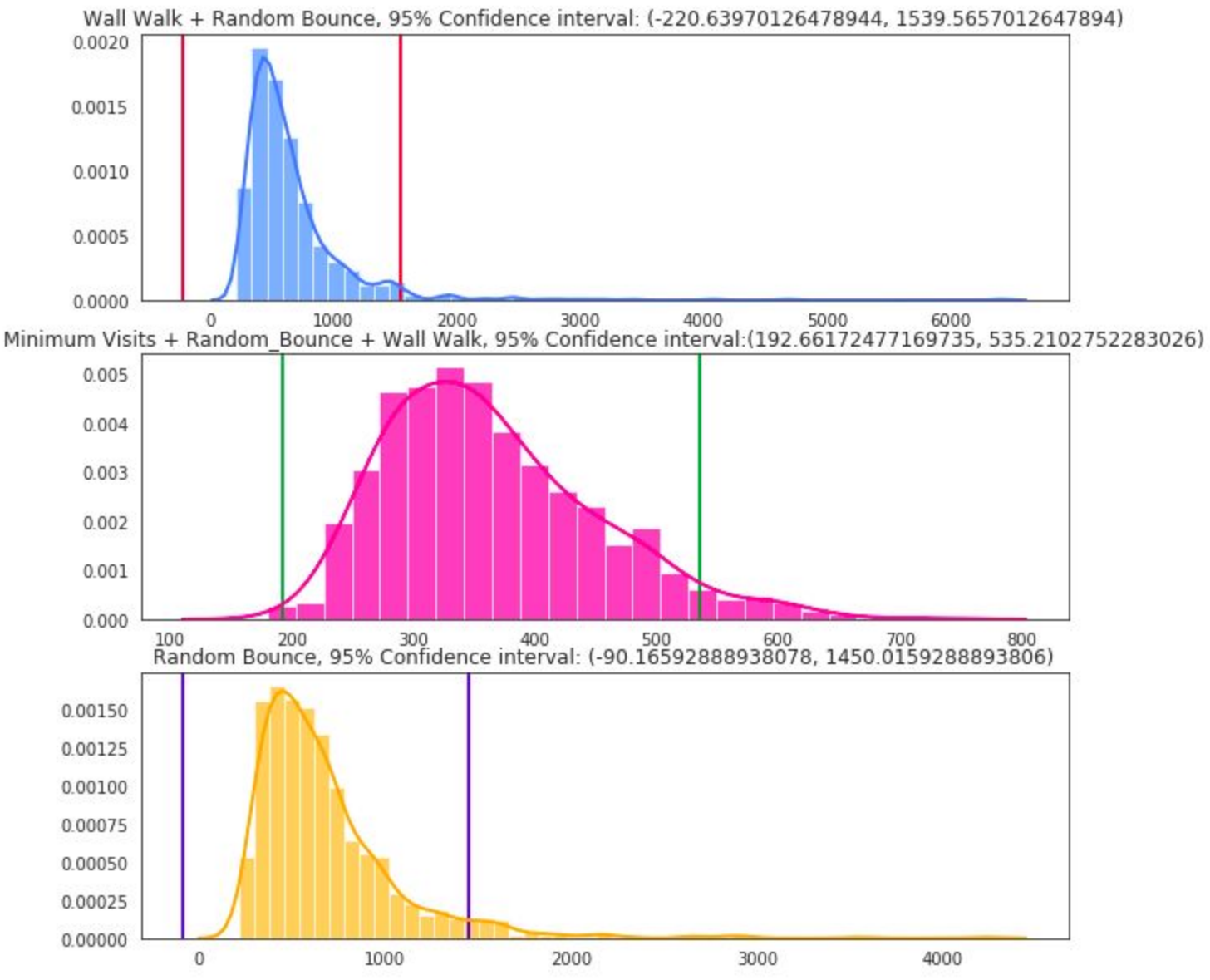
```
random_bounce.append(rb)
wall_walk.append(ww)
minimum_visit.append(mv)
```

Plotting the step data and finding the 95th% confidence intervals for the steps taken, we get:



The x axis says it all! Minimum visits is blistering fast. It gives us the best times of all of them. Clearly some memory improves the performance greatly. Its worst times are the best for random bounce, whose worst times are the best for wall walk.

Now how about combinations of the strategies? I did the same but with the combinations we recorded videos for.



I was very surprised by the outcome in this case. I had assumed the distributions to be roughly normal with no skew and so my automated confidence interval calculator was configured for this. These distributions were all skewed positively to some degree. The negative confidence intervals are **as such not correct due to the assumption that the distribution of the times taken being normal** with no skew. Nonetheless, they give us a good idea of the performance where as expected, the system with all strategies activated outperformed the rest by far. The wall walk seemed to contribute little to the performance given the similarity between the Random bounce and the Random Bounce + Wall Walk strategy.

All the confidence intervals are relatively **highly certain** given how wide they are, especially the Random Bounce one. Tests with bigger rooms just took longer but yielded similar results. **The confidence interval formula** is something that needs review but I stick to it for now given the assumption that the **distributions were normal we had at the start. This however is clearly a weak assumption to some degree because it did not account for the skewness.**

Summary of answers to the project outline.

Describe the situation you are modeling.

Modelling a roomba bot incorporating different cleaning strategies to maximize coverage.

SImulation Goals summary

To clean the whole house. I needed the number of steps of moves it made to clean the whole room given a configuration.

Rules of the strategies summary

In summary:

Wall walk:

Favor moving along a wall as soon as one is detected. Either way, favor cells that have dirt even if they are not along the wall.

Random Bounce:

Follow a specific direction until a wall is hit then switch direction to a random one in the von Neumann neighborhood. Nonetheless, if a dirty cell is encountered along the way, move to it regardless of the direction we are fixed to.

Minimum Visits:

Favor moving to cells that you have visited the least. Nonetheless, favor moving to dirty cells if they are available in the neighborhood.Identify any modeling assumptions and explain under what circumstances these assumptions may and may not be valid.

Parameter description

This has been done extensively in the documentation section above. This has also been emphasized in the code through comments and docstrings

Output description

I have specified this in great detail in the discussion above. The outputs depend on the parameter settings. The different expected results have been discussed in the documentation section.

Output distribution and visualization

This has been done extensively in the Performance Test section where I identified the normalcy and skewness of the distributions and the caveats of this with respect to the confidence intervals. I also evaluated the validity of this gaussian assumption here.

Distribution of steps

The distribution was discussed. There were signs of a gaussian distribution with a heavy positive skew and this was identified.

Best strategy to follow

I identified the best strategy to follow as one with all settings toggled on. This is done after seeing the general performance and even during the coverage analysis as well as the demonstrations.

95% Confidence Intervals

Done in the title of the plots. I also spoke of why the negativity of the skewed results arose and why this came as so because of my distribution assumption.

Confidence intervals certainty

This has been done in the test region.

Simulation refinement.

I don't think this helps, I tried a bigger room and more simulations, it barely changes the intervals, it just smoothens the distribution thus I found 1000 to be enough. I just need a more robust formula for skewed Gaussians when finding the confidence intervals..

Code

The IPNYB has been attached with outputs. Note, the blink demos take a while to run hence the videos. Also, you might want to play those at twice the speed and skip some sections since. Except for the minimum visits, that was fun to watch.

Colab Link here:
https://colab.research.google.com/drive/1Zy2GyqbE7bfpf7oaph0f4dmcC-5o2jAe?authuser=2#scrollTo=Wi76UullXlH1

Github Repo with Snapshot:
https://github.com/GitWahome/Roomba-Bot-simulation