CS 7638: Artificial Intelligence for Robotics

Mars Glider Project

Fall 2019 - Deadline: Monday October 7th, Midnight AOE

Project Description

The goal of this project is to give you practice implementing a particle filter used to localize a robotic glider that does not have access to terrestrial based GPS satellites. The glider is released from a spacecraft over the surface of mars, and receives a distance to ground measurement from a downward facing radar, as well as an altitude estimate from a barometric pressure sensor.

Your glider has an on-board map of the area it is being dropped into. The map covers an area 10 km on a side (100 sq km), and the robot is supposed to be dropped somewhere near (0,0). The map was generated by radar from the mars surveyor satellite mission, and has a 1x1 meter resolution. Dimensions in the map range from -5,000m to 5,000m.

Your glider will exit the reentry craft somewhere within the mapped area hopefully near (0,0), but possibly as far off as +/- 250m in both X and Y. It will eject out of the reentry craft approximately 5,000 meters (+/- 50m) above "sea level".

The orbit of the re-entry craft is designed so that it will enter the Martian atmosphere with a heading of zero radians (due east) with respect to your map, but due to atmospheric turbulence, the actual heading of your glider upon release may deviate from this planned heading based upon a Gaussian distribution with mu=0 and sigma= pi/4.

As mars has a thin atmosphere, your glide ratio is 5:1, which means that the glider moves forward 5 meters for every meter it falls. It moves 5m/sec, and falls 1m/s. This means that you have at least 900 seconds to localize the glider before it potentially goes "off-map" and is lost.

You also have up to 4,500 seconds of "glide time" before the glider risks hitting the surface, but would need to steer the glider to keep it within the boundaries of your known map.

[Note that your software solution is limited to 10 seconds of "real" CPU time, which is different from the simulated "glider time".]

Note that the radar sensor [sense function] gives you a (noisy) distance to "ground", and not "sea level" (on mars, "sea level" is defined as the mean ground height). The barometric sensor [get_height function] gives you the gliders distance above "sea level", modulo the (unknown to you) atmospheric offset due to the naturally changing barometric pressure. The atmospheric offset will be static (unchanging) over the course of a single "drop", but you may need to estimate it to localize the glider successfully.

The glider.py file (which you should not modify, but may examine or import) implements the simulated glider. The opensimplex.py file (which you can safely treat as a black box and should not modify) is used when generating the map function.

The marsglider.py file contains two functions that you must implement, and is the only file you should submit to Canvas.

Part A

The first function is called estimate_next_pos, and must determine the next location of the glider given its atmospheric height and the radar distance to the ground. Note that each time your function is called, you will receive one additional data point, and it is likely you will need to integrate the information from multiple calls to the function before you will be able to correctly estimate your glider's position. The "OTHER" variable is passed into your function and can be used to store data which you would like to have returned back to your function the next time it is called (at the gliders next timestep). Note that in part A you are not able to modify the gliders path, so it will be gliding in a (relatively) straight line.

Part B

The second function is called next_angle. The goal of this function is to set the turn angle of the glider (via the rudder) so that the glider returns to the center of its map (0,0) as it glides down to the ground.

Your two functions will have the same input (barometric altitude and RADAR distance to ground), but the next_angle function will return a turning angle in radians (zero for no turn) instead of a predicted location.

The goal in part B is to return your glider as close to (0,0) as possible in X and Y. Note that you only need to return the glider to (0,0) once as the z (height) decreases to be successful, and the actual z value when this occurs does not matter as long as you have not hit the ground yet, so you may take multiple passes to return the glider to (0,0).

The Map Function

In both parts, your function is provided with a function (called the mapFunc) as an input. You may call this function with a specific (X,Y) location and it will return the elevation of the ground above "sea level" at that location on the map of the Marian surface. (We are using functional programming and the OpenSimplex noise model so that we do not need to maintain and pass around a large data structure that contains all of the map data. You can safely ignore how the map works, and just make use of it.)

Submitting your Assignment

Your submission will consist of the marsglider.py file (only) which will be uploaded to Canvas. Do not archive (zip,tar,etc) it. Your code must be valid python version 2.7 code, and you may use external modules such as numpy, scipy, etc that are present in the Udacity Runaway Robot auto-grader. [Try to ensure that your code is backwards compatible with numpy version '1.13.3' and scipy version '0.19.1']

Your python file must execute NO code when imported. We encourage you to keep any testing code in a separate file that you do not submit. Your code should also NOT display a GUI or Visualization when we import or call your function under test. If we have to manually edit your code to comment out your own testing harness or visualization you will receive a -20 point penalty.

Calculating your score

The test cases are randomly generated, and your particle filter is likely to also perform differently on different runs of the system due to the use of random numbers. Our goal is that you are able to generate a particle filter system that is generally able to solve the test cases, not one that is perfect in every situation.

You will receive seven points for each successful test case, even though there are 20 test cases in total (10 in part A, 10 in part B). This means that you only need to successfully complete 15 of the 20 test cases to receive a full score on this assignment. Your maximum score will be capped at 101, although if you are able to solve more than 15 test cases you can brag about it.

Testing Your Code

NOTE: The test cases in this project are subject to change.

We have provided you a sample of 10 test cases where the first five test cases are EASIER than the actual test cases you will be graded with. Each of these first five test cases are designed to allow you to test one aspect of the simulation. Test cases 6-10 are more representative of the test cases that will be used to grade your project.

We may grade your code with 10 different "secret" test cases that are similar, but not an exact match to any of the publicly provided test cases. These "secret" test cases are generated using the generate_params_marsglider.py file that we have provided to you. You are encouraged to make use of this file to generate additional test cases to test your code.

We have provided a testing suite similar to the one we'll be using for grading the project, which you can use to ensure your code is working correctly. These testing suites are NOT complete as given to you, and you will need to develop other test cases to fully validate your code. We encourage you to share your test cases (only) with other students on Piazza.

You should ensure that your code consistently succeeds on each of the given test cases as well as on a wide range of other test cases of your own design, as we will only run your code once per graded test case. For each test case, your code must complete execution within the proscribed time limit (10 seconds) or it will receive no credit. Note that the grading machine is relatively low powered, so you may want to set your local time limit to 5 seconds to ensure that you don't go past the CPU limit.

We are using the bonnie autograder system which allows you to upload and grade your assignment with a remote / online autograder. See the submit.py file posted as part of the assignment on Canvas for details. You are not required to use this online/autograder feature, but if you do, it will give you more assurance that your code will work correctly with our autograder when we grade the files you submit on Canvas. We **may also choose to use the last grade you receive via the remote autograder as your final grade** at our discretion. (See the "Online Grading" section of the Syllabus.)

Academic Integrity

You must write the code for this project alone. While you may make limited usage of outside resources, keep in mind that you must cite any such resources you use in your work (for example, you should use comments to denote a snippet of code obtained from StackOverflow, lecture videos, etc).

You must not use anybody else's code for this project in your work. We will use code-similarity detection software to identify suspicious code, and we will refer any potential incidents to the Office of Student Integrity for investigation. Moreover, you must not post your work on a publicly accessible repository; this could also result in an Honor Code violation [if another student turns in your code]. (Consider using the GT provided Github repository or a repo such as Bitbucket that doesn't default to public sharing.)

Frequently Asked Questions (F.A.Q.)

Q How can I simplify this problem to make thinking about it easier?
A Try solving it in only 2 dimensions, and take a look at the video

that inspired the problem: Particle Filter explained without equations - https://www.youtube.com/watch?v=aUkBa1zMKv4

Q Are you SURE this can be solved using a particle filter? *A* Yes. See https://www.youtube.com/watch?v=RvhPd16Th2I

Q How exactly are my sources of data related? *A* Radar gives the glider's height over ground (+/- measurement error), the barometric measurement height gives the gliders height above sea level (+/- atmospheric offset, if any), and the mapFunction returns the elevation of the ground (above/below sea level) at a specific (x,y) location.