CS 7638 - Robotics: AI Techniques - Meteorites Project

Summer 2023 - Due Monday, June 5th, Midnight AOE

Table of Contents

Introduction

In this project, Earth is threatened by a shower of meteorites falling in your location. It is your task to receive sensor readings of the locations of these meteorites, predict where each of the meteorites will be one tenth of a second later using Kalman Filters (KFs), and finally, destroy each meteorite before it hits the ground by firing your laser turret at it.

This project consists of two parts:

1. Estimation: Estimate where meteorites will be one timestep into the future
   a. Estimation of a small number of meteorites' positions with no noise in the observations—5% of grade
   b. Estimation of the positions of many meteorites given noisy measurements—75% of grade
2. Defense: Aim and fire your laser turret at incoming meteorites before they hit the ground—20% of grade

Submitting Your Assignment

Your submission will consist of ONLY the `turret.py` file, which you will upload to Gradescope.

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). For an example of this, note how the author of this project's code cited the source for the `clamp` function in `runner.py`.

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 server for your repository, or a git server such as Bitbucket that does not default to public sharing.)

Project Description

The motion model of the meteorites takes the form

$$x(t) = c_{pos_x} + c_{vel_x} t + \frac{1}{2} S_{acc} c_{acc} t^2$$

for the meteorite's x-position, and

$$y(t) = c_{pos_y} + c_{vel_y}t + \frac{1}{2}c_{acc}t^2$$

for its y-position. $S_{acc} = \frac{1}{3}$ is a constant.

Time is delimited in discrete steps (t = 0.0, 0.1, 0.2, ...). Each timestep is $dt = 0.1$ seconds in duration. Each meteorite's motion can be modeled using x, y, dx, dy, a. a is acceleration; note that, due to how the acceleration term is defined, the x- and y-components of a meteorite's motion are correlated!

In most parts of this project, your turret's observations of the meteorites' positions are noisy, so you will leverage the uncertainty-handling properties of Kalman Filters to predict their positions more precisely.

Environment:

In this project, your world is a 3-by-2 rectangle, with the X-range [-1, 1] and Y-range [-1, 2]; (-1, -1) is the lower left corner, and your turret is located at (0, -1), with y = -1 being the ground. This coordinate system is used throughout this project to define all entity locations. The laser turret's aim angle is 0.0 rad when the laser is pointed along the ground to the right, and π rad when the laser points along the ground to the left.

HINT: On line 21 of `turtle_display.py`, ensure the `DEBUG_DISPLAY` variable is set to `True` to show meteorite IDs and the (x, y) coordinates of the corners of the world in the GUI.

Estimation

Estimation Part (a)

In this part of the project, you are predicting the location of a small number of meteorites one timestep into the future given the meteorites' current true positions. That is, the measurement information (`meteorite_observations`) provided to the `predict_from_observations` function in Part (a) of the Estimation part of this project is *NOT* noisy.

Inputs: The `predict_from_observations` function takes in a tuple of tuples of meteorite ID numbers, x-coordinate observations, and y-coordinate observations; that is, the `meteorite_observations` argument has the form

```
((0, -0.83, 0.46),
(1, 0.44,  0.8),
(3, -0.72, -0.3),
...
(1003, 0.34, 0.1))
```

Note that the meteorites in `meteorite_observations` are not guaranteed to be sorted in any sort of order, so do not expect the ID numbers to be sequential.

Outputs: The output of the `predict_from_observations` function should be a tuple of tuples of estimated meteorite locations one timestep into the future (i.e. the inputs are for measurements taken at time t, and you return where the meteorites will be at time t+1). This output should be provided in the same format as the input, `meteorite_observations`. Note that the ordering of the components in each meteorite's tuple should be `(meteorite ID, x-coordinate, y-coordinate)`.

Goal: To get full credit for this part of the estimation part of the project, your `predict_from_observations` function will need to provide "close enough" (within 0.01 units of distance of each meteorite's true position) predictions of the meteorites' positions for 75 consecutive timesteps before 400 timesteps have elapsed.

The meteorites in case0.json are not subject to acceleration, whereas the meteorites in case1.json are subject to acceleration.

Due to the lack of noise on the meteorite observations in this part of the project, it is possible to pass this part of the project using only the prediction step of the Kalman Filter and ignoring the measurement update step. As for why this could cause problems in the remainder of the project, please see the Why bother implementing a Kalman Filter for the non-noisy Estimation part of the project? question in the FAQ.

NOTE: Due to the ordering of steps of the simulation in runner.py, when considering the Prediction and Observation steps of the KF, for this project, perform the Observation step before you perform the Prediction step.

How To Test Your Part (a) Estimation Code To test your code on an estimation case and see a visualization of the simulation, run the following in your Python environment (the `case` argument may be 0 or 1; the command to run case 0 is shown here):

```
python test_one.py --case 0 --display turtle kf_nonoise
```

This will run a simulation with a a visualization similar to that shown in "How To Test Your Part (b) Estimation Code."

A similar command lets you run the test with only text output (no visualization). This text-only mode is what `test_all.py` uses. (see "Testing Everything" below)

```
python test_one.py --case 0 --display text kf_nonoise
```

As before, the `case` argument may be 0 or 1.

Notes on testing your Part (a) Estimation Code This part of the project is meant to help you verify that your KF works correctly in simple scenarios before applying it to more complex scenarios. Passing this part of the project *does not* guarantee that your KF is set up correctly, but case0.json and case1.json are small enough test cases that you can work through a timestep or two for one or more meteorites by hand while stepping through your code with a debugger to verify that your KF implementation is consistent with your hand calculations. If your code passes these no-noise cases but cannot pass the noisy cases in Estimation Part (b), we recommend verifying your KF implementation by changing the `noise_sigma_x` and `noise_sigma_y` in your local case0.json and case1.json test cases and working through a timestep or two of the simulation by hand to identify where your KF's handling of noise may be incorrect.

Estimation Part (b)

In this part of the project, you will be estimating the location of each meteorite visible on the screen one timestep in the future given *noisy* measurements of meteorite locations you have for the current timestep (`meteorite_observations`).

Inputs: Same as in Estimation Part (a), but this time, the x- and y-components of the measurements provided in the `meteorite_observations` argument are *noisy*.

Outputs: Same as in Estimation Part (a).

Goal: To get full credit for this part of the project, your `predict_from_observations` function will need to provide "close enough" (within 0.02 units of distance of the true position) position predictions of at least 90% of the meteorites within the 3-by-2 box for five (5) consecutive timesteps. This must be accomplished within 400 timesteps (40 seconds) (and on Gradescope or when using `test_all.py`, within 10 real-world "wall-time" seconds). Passing back predictions for non-existent meteorites (e.g. meteorites that have hit the ground and have an ID of -1) will not affect your score.

NOTE: Due to the ordering of steps of the simulation in runner.py, when considering the Prediction and Observation steps of the KF, for this project, perform the Observation step before you perform the Prediction step.

How To Test Your Part (b) Estimation Code To test your code on an estimation case and see a visualization of the simulation, run the following in your Python environment (the `case` argument may be 2-17; the command to run case 2 is shown here):

```
python test_one.py --case 2 --display turtle estimate
```

When you run a case with the `turtle` visualization option, you should see something like what is shown in the image below. The gray circles represent the actual locations of meteorites. A red dot indicates a prediction that is too far from the meteorite's actual location to count as correct, and a green dot indicates an estimate close enough to be counted as correct.

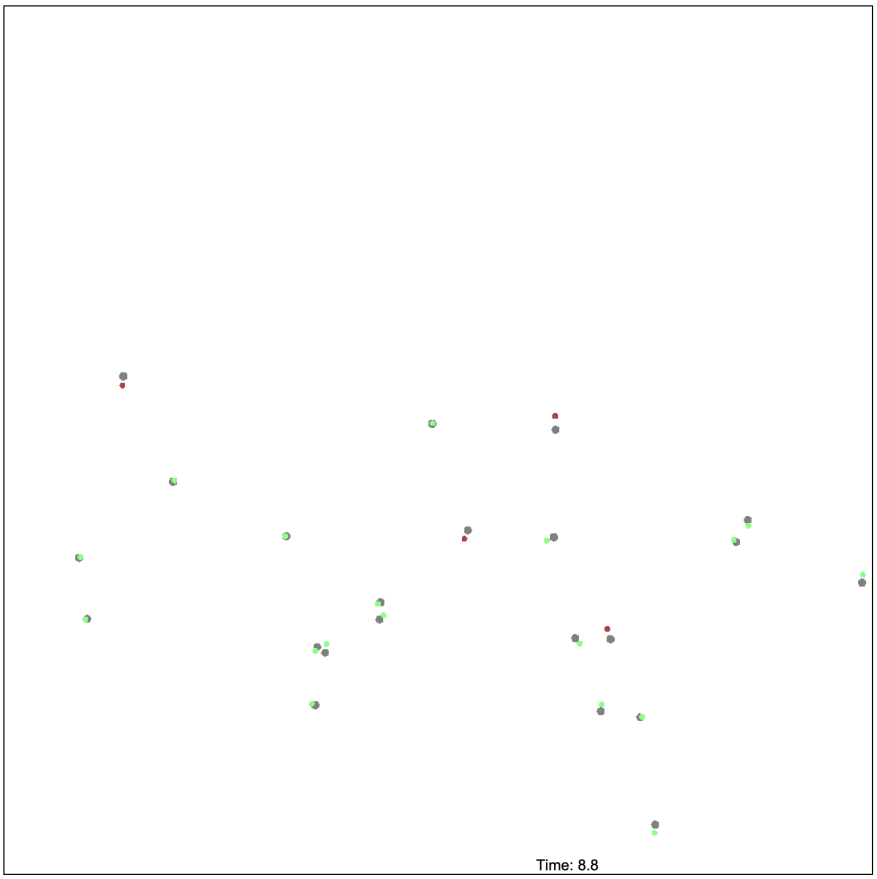

Figure 1: Estimation visualization

A similar command lets you run the test with only text output (no visualization). This text-only mode is what `test_all.py` uses. (see "Testing Everything" below)

```
python test_one.py --case 2 --display text estimate
```

As before, the `case` argument may be 2-17.

Defense

For the defense part of the project, you will be devising a simple algorithm to aim and fire your laser turret at falling meteorites. The defense part of the project makes use of the predictions of the meteorite locations computed by `predict_from_observations` in the Estimation Part (a) portion of the project. *(HINT: Don't over-think your strategy here; perhaps simply aiming at the lowest meteorite above some minimum threshold is sufficient!)* A meteorite is destroyed with probability 0.75 if the laser line comes within a small distance

(the value denoted as `min_dist` in the relevant case file) of it. When the laser fires at time t, the shot hits the meteorite at time t+1. The laser line itself is 1.1 units of length long, measured from the turret. The laser can only fire a limited number of shots before it runs out of power; the number of shots remaining are displayed in the GUI or command line output. The turret's position is fixed at an x-position of 0 and a y-position of -1; the turret may rotate, but not change its x-y position.

Each meteorite's ID number is unique as long as the meteorite has not been destroyed. When a meteorite is destroyed, its ID number is set to -1. This ID number change is handled by the simulation. Keep in mind that you may want your turret to check that it does not try to aim at a meteorite with an ID of -1!

Inputs:

This function takes in a float corresponding to the laser turret's current aim angle, in radians.

Outputs:

The output of this function is a tuple containing two values:

- Float: The change in aim angle (in radians) you want the laser to execute; if the magnitude of this value is greater than `max_angle_change` (0.0873 rad; approximately 5 degrees), it will be lowered to `max_angle_change` rad, but with the sign of the angle you outputted.
- Boolean: True if the laser should fire the next timestep; False, otherwise.

Thus, the laser turret can both rotate and fire in the same timestep, if desired. Note that the float should be returned first and the Boolean second (and that enclosing parentheses are optional when returning a tuple from a function).

Also note that trying to move the laser's aim outside of the [0, π] range will result in its aim being clamped to 0 or π, respectively. The laser's aim angle does NOT wrap around—if you output an angle change that would set the laser's current aim to, say, 3.3 rad, the laser's aim will stay π rad until you change the laser's aim back to within the [0, π] range.

Goal:

Your goal in the defense part of the project is to make sure your laser turret survives for 400 timesteps. Your laser turret starts with a specific number of health points (HP), which are shown below the turret in `turtle` simulation mode and printed to the command line in `text` mode. Each time a meteorite hits the turret or the ground (y = -1), the turret loses one HP. Credit is given for a case if the turret's HP is 1 or greater by the end of the 400-timestep bout (on Gradescope and in `test_all.py`, there is also a 45-second wall-time time limit); no credit is given if the turret's HP drops to 0 within that time limit.

How To Test Your Defense Code

To test your code on a defense case and see a visualization of the simulation, run the following in your Python environment (the `case` argument may be 2-17):

```
python test_one.py --case 2 --display turtle defense
```

When you run the above command, you should see something like the image below.

A similar command lets you run the test with only text output (no visualization); this is the mode that `test_all.py` uses to run all test cases. (See "Testing Everything" below) As before, the `case` argument may be 2-17.

```
python test_one.py --case 2 --display text defense
```

Testing Everything

To test all of the local estimate and defense cases using the `text` display option, use the command

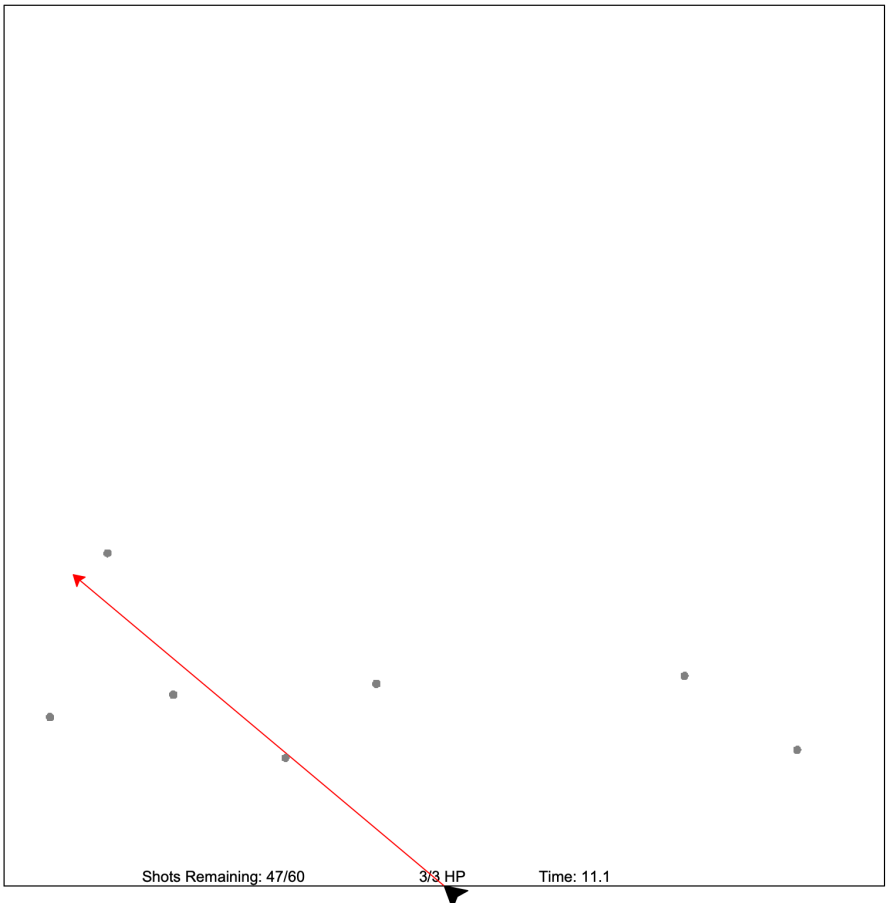

Figure 2: Defense visualization

```
python test_all.py
```

This is the testing mode used by Gradescope.

Generating New Test Cases

The cases used for grading on Gradescope are similar to those provided to you, but not the same. You can use `generate_test_case.py` to generate additional test cases to more rigorously test your code, To see all of the command line arguments for the `generate_test_case.py` script, run the following in your Python environment:

```
python generate_test_case.py --help
```

To create a new case, run as follows:

```
python generate_test_case.py my_case [additional arguments here]
```

To use this new test case, pass the filename to `test_one.py` using the `--case` argument:

```
python test_one.py --case my_case --display turtle defense
```

Note: The new case files are not included in the cases executed by `test_all.py`.

Frequently Asked Questions

- How do Kalman Filters apply in this project?
- How do I share data between functions in my Turret class?
- Do I need to simulate the motion of the meteorites myself?
- Why bother implementing a Kalman Filter for the non-noisy Estimation part of the project?
- Do I need an R matrix in the non-noisy Estimation part of the project?
- Should I change my R matrix depending on which case is being run?
- Do I need a Q matrix in this project? Do I need the u vector in this project?
- Why do I get less credit on Gradescope than I do on my local machine?
- How do the Gradescope cases and the cases we have differ?
- Further questions?

How do Kalman Filters apply in this project?

We know the structure of the motion model that governs the motion of the meteorites, but each meteorite has different coefficients in its equation of motion, which means we can't just apply the motion model to predict a particular meteorite's next location. Kalman Filters allow us to combine our knowledge of the motion model's structure and our estimate of our uncertainty of each element in the state of a particular meteorite with observations of the meteorite's positions over time to predict where the meteorite will be at a future time. Since each meteorite has its own motion model coefficients and therefore moves slightly differently than all the other meteorites, we need one Kalman filter for each meteorite. You'll want to create and update separate $\bar{x}$s and Ps for each meteorite, using the Kalman filter equations. The state transition matrix (aka motion model matrix, F), measurement model matrix (H), and observation uncertainty matrix (R) are constant and the same for all meteorites.

Hint: Many students use a Python dictionary data structure to keep track of the state estimates and P matrices for each meteorite. However you choose to keep track of your state estimates and P matrices, remember that your turret.py's internal storage for $\bar{x}$, P, or whatever else you create in turret.py are not automatically updated when a meteorite hits the ground or is destroyed! Keep that in mind if your turret wastes its laser shots firing at nothing.

How do I share data between functions in my Turret class?

In your implementation of the `Turret` class, you can refer to the current Turret instance using `self` and attach additional data to it. Here is an example of creating a `value` variable in a `Counter` class that can be used in other functions in the `Counter` class:

```
class Counter(object):

    def __init__(self):
        self.value = 0

    def increment(self):
        self.value += 1

    def show(self):
        print(self.value)

ctr = Counter()
ctr.increment()
ctr.increment()
ctr.show()        # should display '2'
```

Do I need to simulate the motion of the meteorites myself?

No, the meteorites' motion is already taken care of in `runner.py`; your turret needs to use observations of where meteorites are to predict where they will be one timestep into the future (and, in the defense portion, fire a laser at their predicted locations). When predicting meteorite location estimates, your turret only needs to store some data from the previous timestep (t-1) and use the `meteorite_observations` (meteorite location measurements at time t) provided to `predict_from_observations` to predict where meteorites will be at the next timestep (t+1).

Why bother implementing a Kalman Filter for the non-noisy Estimation part of the project?

Estimation Part (a) of this project is designed specifically to allow you to test your KF in very simple scenarios to ensure that it works before moving on to the remaining parts of the project. Use Estimation Part (a) to debug your KF implementation and your handling of multiple meteorites' KF components before moving on to the cases with noise in Estimation Part (b).

Do I need an R matrix in the non-noisy Estimation part of the project?

Yes, you need an R matrix with positive, nonzero values on the diagonal for your KF implementation to work in Estimation Part (a). If your code for Estimation Part (a) gets full credit but does not have an R matrix, you are likely only performing the prediction step of the KF algorithm instead of both the prediction and measurement update steps.

Should I change my R matrix depending on which case is being run?

No, we intentionally designed this project such that one R matrix can be used across all cases. We recommend looking at the noise standard deviations `noise_sigma_x` and `noise_sigma_y` in cases 2-17 when selecting your values for R.

Do I need a Q matrix in this project? Do I need the u vector in this project?

No, the Q matrix (model uncertainty matrix) and u vector (external force vector) are not necessary for this project.

Why do I get less credit on Gradescope than I do on my local machine?

Keep in mind that (1) Gradescope uses different case files than what you have access to (though they are very similar), and (2) your local computer may be faster than what Gradescope uses to run your code. If your code times out on Gradescope (`execution_time_exceeded`), think about whether there is a way you can make your code more efficient. Additionally, when running `test_one.py` for an `estimate` or `kf_nonoise` case, there is no

wall-time time limit applied—just a limit to the maximum timesteps your solution may take to converge its estimates.

How do the Gradescope cases and the cases we have differ?

The cases you have access to locally are representative of the cases on Gradescope. None of these cases–neither the ones you have access to, nor the ones on Gradescope–are meant to be unsolvable. All of the cases are generated with the same script and with similar, but not identical, parameters. There are more grading cases on Gradescope so that you can still get a good grade on the project even if your turret.py doesn't pass a couple of cases.

Further questions?

If you have additional questions, first, please check the course FAQ document, syllabus, policy guidelines, and code review guide, which can be found on Canvas > Files. Also try searching Ed Discussions for keywords related to your problem. If none of these answer your question, please feel free to ask on Ed Discussions in accordance with the guidance in the code review document. (Remember: if you are posting code in your Ed Discussions post, your post must be *private*!)