ENGR 3H Final: The Great ENGR 3/10H Triathlon (Parts 1 and 2: The Biathlon: Skiing and Shooting)

Tim Matchen

May 2019

1 Introduction

For your final project, you have been subdivided into teams and are being given three distinct challenges in which to demonstrate your mastery of the material. Each challenge will be scored separately, and a bronze, silver, and gold medal will be awarded. (Note: Not actual medals. Still working on a grad student budget here, people). These challenges will cover neural networks, Q-Learning, and Monte Carlo Tree Search (coming soon!). Each task is detailed below.

2 The Great Slalom

The Goal: Reach the bottom of the hill in as little time as possible while tagging every gate along the way! Penalty time will be added for missed gates. The fastest time to the bottom wins!

Requirements: You will need to write three programs for the Great Slalom:

1. An ODE file called `slalom.m`;
2. A function `slalom_train.m` that takes as an input a set of x-y coordinates for flags (explained below) and carries out Q-Learning; and
3. A script `slalom_run.m` that uses the learned Q-Table to run the slalom course.

The specifics of each program are described below.

slalom.m The ODE for this problem consists of a trajectory defined in terms of a constant speed with changing direction. In other words:

$$v_x^2 + v_y^2 = v_{max}^2, \tag{1}$$

where v_{max} is a constant. You will be simulating this ODE using a time step of 1 second at a time; over that time step the value of v_x will shift from its current value to its new value (the chosen action!) at a constant rate. Therefore, over the 1-second window, the value of the velocity in the x-direction is given by:

$$v_x(t) = v_{x_0} + t\frac{dv_x}{dt}, \tag{2}$$

where $\frac{dv_x}{dt}$ is constant over the 1-second interval. v_y should be defined relative to v_x. Additionally, note that v_y is **negative**. You're going downhill, after all.

slalom_train.m As this is a slalom course, the goal is not simply to reach the bottom of the hill, but to pass through a series of gates on the course. Missing a gate incurs a significant penalty, sufficient that it is highly unlikely the racer can win against a racer who does not miss a gate. The race starts at $(0, 0)$ and ends when the racer crosses $y = -10$. The course does not extend infinitely in the x-direction; if the racer reaches $|x| > 2.5$, they are disqualified. Each gate is a square with side length 1 centered on an integer vertex, such as $(2, -2)$, $(3, -5)$, etc. To get credit for passing through the gate, the racer must enter this square. For example, to pass through a gate located at $(2, -2)$, the racer must enter the square defined by the vertices $(1.5, -1.5)$, $(1.5, -2.5)$, $(2.5, -1.5)$, and $(2.5, -2.5)$. The list of gates will be given as an $N \times 2$ array of the x- and y-coordinates of the gates' centers and will serve as the input to `slalom_train.m`.

At each second of the game, the racer chooses between three actions: whether to end the 1-second interval with $v_x = -v_{max}$, $v_x = 0$, or $v_x = v_{max}$. The additional specifics of the problem, including rewards and penalties for different states, are left to you to design. Remember that to implement Q-Learning, you'll need to discretize your state space. *(Hint: MATLAB's built-in round function rounds numbers to the nearest integer.)*

slalom_main.m In this file, you should take the Q-Table you generated in your training function and use it to simulate a full slalom run. Plot the resulting x vs y plot (see Fig. 1 for an example).

3 Bullseye!

The Goal: Successfully fire an arrow to hit a prescribed target up to 200 meters away. The closer you get to the bullseye, the more points! Oh, and there are a bunch of floating orbs you'll need to avoid, too.

Requirements: There is only one function you have to write for this one: `shootyourshot.m`, which takes two inputs, the x-coordinate of the bullseye and a 3×2 matrix of x- and y-coordinates for three floating spheres you must avoid hitting. Getting this function to work, however, will involve several substeps.

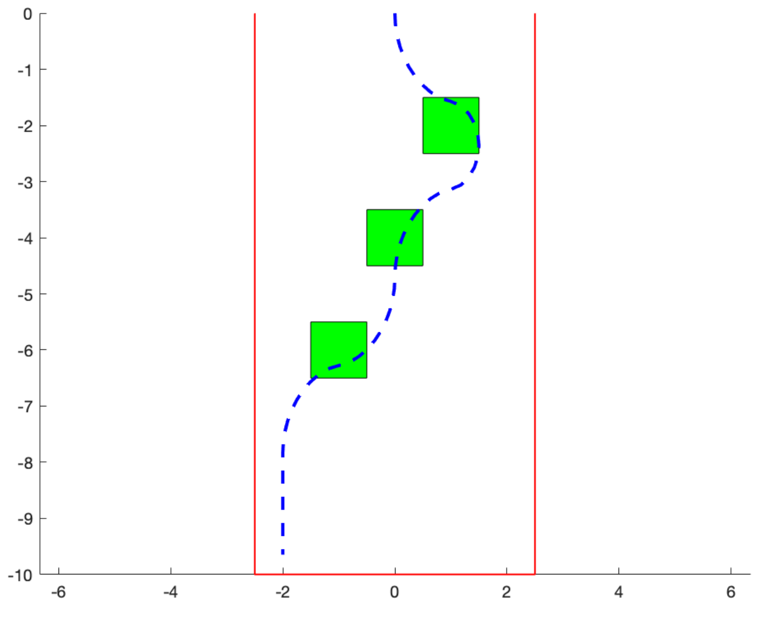

Figure 1: A successful run with $v_{max} = 2$!

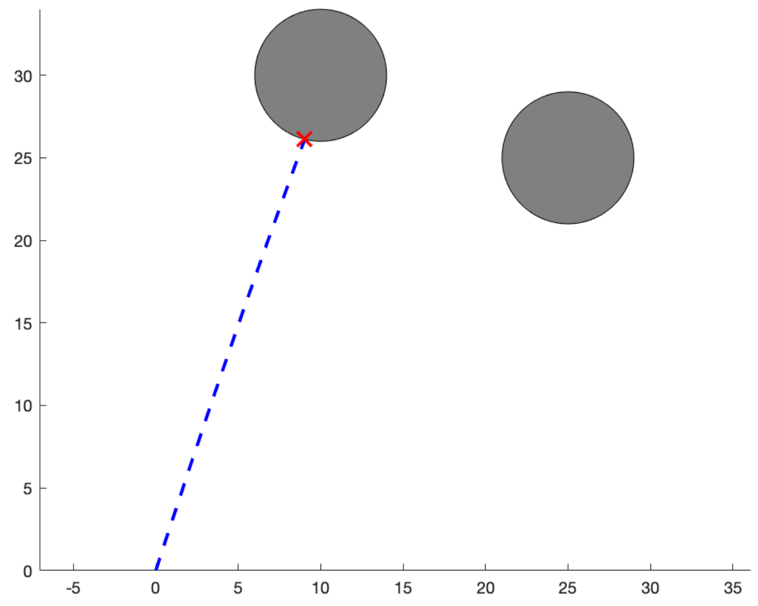

Figure 2: Pictured: A bad shot.

This problem is all about building your own data set on which to train a neural network (or two). Your archer stands at $(0, 0)$ and attempts to hit a target at $(x_{targ}, 0)$. The archer can aim at any angle between 0 and $\frac{pi}{2}$ radians, and the initial velocity of the arrow can range from 0 to 50 m/s. The score the archer receives for landing an arrow at $(x_{hit}, 0)$ is given by:

$$S = 100 \exp - \frac{|x_{hit} - x_{targ}|}{50}; \tag{3}$$

arrows that hit an obstacle instead receive 0 points. Each obstacle has radius 4 (see Fig. 2). You will need to build a neural network to predict the outcome of firing an arrow with a given initial velocity and angle, then utilize that network to choose the combination of power and angle that you believe will generate a shot that lands closest to the target. The target can range from 50 all the way to 200; here, we are assuming the only force acting on our system is gravity, so our equations are:

$$a_x = 0, \; a_y = -9.8. \tag{4}$$

Note it might be easier to build two separate neural networks: one that classifies arrows as either hitting an obstacle or not, and one that makes a regression estimation of where an arrow lands. If you are unsure how to implement code so that your ODE stops when hitting an obstacle (or the ground), look into how

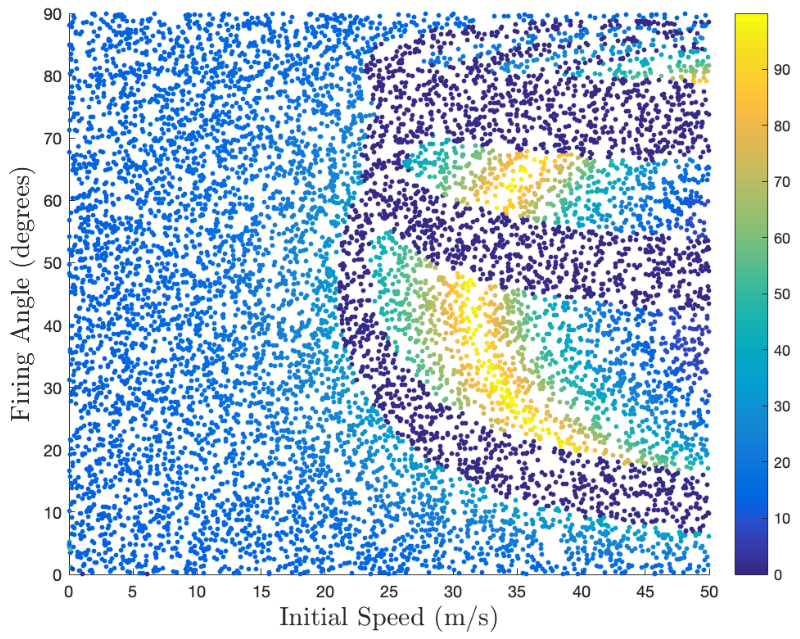

Figure 3: Scores for an example set of obstacles and with the target at $x = 100$. Hopefully you can tell which ones hit the obstacles...

to write an event finder in MATLAB for ODEs. You may also wish to explore other ODE options, such as MaxStep, to ensure the event finder works properly.