CS100R Assignment 5

out: October 29, 2007.
due: **November 7th, 2007.**

1 Introduction

In lecture, we have considered the least squares technique in some detail. In this assignment, you will examine its implementation by optimizing parameters using various search algorithms.

You may wonder why we would want to go to such pains to write effective search algorithms for minimizing the parameters of a line when a closed-form solution exists. As it turns out, writing a closed form solution is often impractical at best, and often impossible. Moreover, the search algorithms you will use in this assignment are very important ones in Computer Science, and they are useful in many other contexts.

2 Least Squares and Variants

Least squares, in general, attempts to find a function which closely approximates a set of data by minimizing squares of the differences between points generated by the function and actual data points. These differences are called *residuals*. While the potential applications of least squares are quite numerous, we are primarily concerned with its utility in finding a line of best fit.

To do this, we will consider a function that describes a line in slope- intercept format: $f(x) = mx + b$. Since we're using the least-squares technique, we'd like to find the values of m and b that minimize the sum of the squares of the residuals. That is, we want to find m and b such that for the set of data points (x_i, y_i) we minimize

$$S(m, b) = \sum_i \left(y_i - (mx_i + b)\right)^2 .$$

Consider the nature of S: it only has one minimum! If we could stand on a graph of S versus m and b, we could roll a ball starting from anywhere and it would eventually end up at the minimum. This means that if we move towards the minimum, we won't get caught in any traps on the way; however, as we've noted in lecture, this doesn't necessarily mean that we can reach the minimum in two steps. With this in mind, we present two methods for finding the best m and b.

2.1 Hill climbing search

First, try finding the best parameters using *hill-climbing*: start at a point and step in whatever direction tends to decrease S. You can start at any point (m, b) and evaluate $S(m, b)$. Check to see how the value of S changes by evaluating $S(m - \epsilon, b)$, $S(m + \epsilon, b)$, $S(m, b + \epsilon)$, $S(m, b - \epsilon)$ and seeing if moving in any direction causes S to decrease. If so, move in that direction. If not, then you've found a minimum.

The key ambiguity, of course, concerns what we mean by "moving a little bit". This is up to you!

2.2 Golden section search

Since evaluating the derivative requires at least two function evaluations, binary search can slow down quite dramatically if the function in question is complicated. But, if we're more careful, we can reduce the number of function evaluations significantly by eliminating the derivative from the picture. Consider what we can tell about a convex function if we have its value at four points: if a minimum or a maximum lies between the first point and the fourth, we can look at the values of the points, remove the first or the last point, and thereby narrow down the interval we're considering (see Fig. 1).

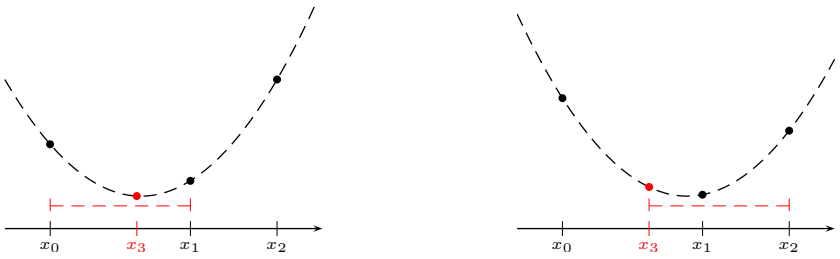

Figure 1: Choosing a subinterval.

If we take the three resultant points, pick a fourth point in their interval, and compute its value, we can repeat the process to obtain an even smaller interval. However, we'd like for the proportions of the intervals to be consistent, so we have to choose carefully.

Let's say we have an interval with three points x_0, x_1, and x_2. Let a be the distance between x_0 and x_1, and b be the distance between x_1 and x_2 (See Fig. 2). We'd like to choose a point x_3 such that the distance c from x_3 to x_1 has the same proportionality to b and to $a - c$ that b has to a. That is,

$$\frac{c}{b} = \frac{c}{a-c} = \frac{b}{a}.$$

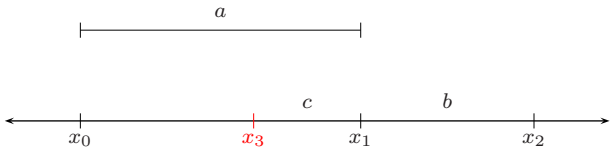

Figure 2: Choosing x_3.

If we note that $c = \frac{b^2}{a}$, we can rework the expression as follows:

$$\begin{aligned} \frac{a - \frac{b^2}{a}}{\frac{b^2}{a}} &= \frac{a}{b} \\ \frac{a^2}{b^2} - 1 &= \frac{a}{b}. \end{aligned}$$

This is a quadratic equation in $\frac{a}{b}$ which has the positive solution

$$\frac{a}{b} = \frac{1 + \sqrt{5}}{2} = \phi = 1.61803\ldots$$

That is, x_0, x_1, and x_2 must form a *golden section*, and if we pick x_3 carefully depending on the orientation of the interval, this will always be true. Specifically, when $x_2 - x_1 > x_1 - x_0$, pick $x_3 = x_1 + (x_1 - x_0)/\phi$; otherwise, pick $x_3 = x_1 - (x_2 - x_1)/\phi$.

There's one other small matter: finding an initial interval. In our binary search, we picked an initial interval by taking some interval and doubling its size until it was valid. Here, we can do the same thing, but instead of doubling, we can increase the interval's size by shifting the middle point to the left or right end and adding a new point beyond the interval in that direction. Again, if we choose these carefully, we can preserve the proportions of the sections and minimize function evaluations. Specifically, when increasing the interval leftward, set $x_1 = x_0$, and then set $x_0 = x_1 - \phi(x_2 - x_1)$; otherwise, set $x_1 = x_2$, and then set $x_2 = x_1 + \phi(x_1 - x_0)$.

Now we have everything we need to write the two parts of this search: finding a valid interval, and reducing the interval.

1. Finding a valid interval is fairly straightforward. Given any starting interval, we simply need to add or remove points in the correct proportions until the value of the function at x_{mid} is less than the values at x_{lo} and x_{hi}. Once we're done, we should have an interval whose points are proportioned by the golden ratio.

2. Subdividing an interval is complicated somewhat by the fact that it may have one of two orientations, depending on whether x_{mid} is closer to x_{lo} or x_{hi}. But, if we check for this, the logic is fairly simple: check a new point inside the interval and, depending on whether its value is greater than or less than the value at x_{mid}, add x_{new} to the interval and drop one of its other points.

Now we can apply these to steps to minimize both m and b: start with a valid interval and repeatedly subdivide it until the interval length is less than some small value ϵ. Again, the choice of this small value is up to you.

2.3 Robot Speedometer

Now that we have seen some search algorithms, let's motivate them with a real-world example: estimating the speed of a robot. We suspect that the robots we use in lab are disobeying orders and lying to us about their speed[1]. You can verify this by issuing a drive command to the robot that tells both wheel motors to drive forward at 50 millimeters per second:

```
>> robotDriveDirect(X, 50, 50)
```

While it's driving, you can repeatedly poll the distance sensor on the robot to see how far it's gone:

```
>> robotGetSensors(X, 'drive.distance')
```

If you also get time values for each of these samples using `tic` and `toc`, you'll have a set of points which, if the robot were indeed driving at a constant velocity and telling the truth, should lie on a line with slope equal to 50 millimeters per second[2].

Write a function to obtain this robot distance data (remember that the `robotDriveDirect` command will tell the robot's wheels to drive at constant velocity until you send another command that says to do otherwise) and return it as an $n \times 2$ array. You can use the `tic` and `toc` functions in a `for` loop if you like, or, if you're up for an adventure, try using Matlab's `timer` object.

Also write a function which creates a random test dataset. It should pick values for m and b randomly (their ranges are up to you) and produce a set of points that lie on the line. But then it should take each point and move it by a random distance to fake the presence of error in the data. You can use this function to test your line-fitting code when a robot isn't available.

[1] This is probably because we've put far more weight on them than they were designed to lug around.

[2] Since the `drive.distance` sensor returns the number of millimeters the robot has travelled since it was turned on, the intercept of the line will represent the number of millimeters the robot travelled between being powered on and the first time you queried the sensor.

3 What to hand in

1. Create two functions which return data sets: `getdata_robot` and `getdata_test`. They should operate as described in Section 2.3.

2. Create a function `plot_line` which accepts a set of points, a slope, and an intercept. The function should plot the points and the line. We will go over Matlab's `plot` command briefly in section.

3. Create a function `ls_hill` which finds a best-fit line for a set of points using hill-climbing to minimize the least-squares residuals. In each iteration of the search, call your `plot_line` function to display the current progress. You'll need to use the `drawnow` command to force Matlab to update the figure window, and you may want to use the `pause` function if things are running too fast for you to see what's going on.

4. Create a function `ls_golden` which finds a best-fit line using golden-section search to minimize the least-squares residuals, using the `plot_line` function as before.

5. Write versions of `ls_hill` and `ls_golden` which minimize the total least-squares residuals, as discussed in class.