16-811: Math Fundamentals for Robotics, Fall 2018
Assignment 4

DUE: Thursday, October 25, 2018

1. Consider the following differential equation over the interval $[1, 2]$:

$$\frac{dy}{dx} = \frac{1}{3y^2}, \qquad \text{with } y(2) = 1.$$

(Caution: The "initial condition" is specified at the right endpoint in this problem.)

(a) Obtain an exact analytic solution $y(x)$ to this differential equation.

(b) Implement and use Euler's method to solve the differential equation numerically. Use a step size of 0.05. How accurate is your numerical solution? (Compare the numerical solution to the exact solution, perhaps as follows: Create a table with one row for each x_i encountered using the given step size. The row might mention x_i, the true value $y(x_i)$, the estimated value y_i computed using Euler's method, and the error $y(x_i) - y_i$. Or report similar results in some other way. You might even want to combine all the methods of this problem into one big table. Graphing the results is also a good idea. Reporting both graphs and tables is perhaps the best thing to do.)

(c) Implement and use a fourth-order Runge-Kutta method to solve the differential equation numerically. Again, use a step size of 0.05. Again, how accurate is your numerical solution?

(d) Finally, implement and use fourth-order Adams-Bashforth for the differential equation. Again, use a step size of 0.05. Initialize the iteration with the following four values:

$$\begin{aligned} y(2.15) &= 1.04768955317165, \\ y(2.10) &= 1.03228011545637, \\ y(2.05) &= 1.01639635681485, \\ y(2) &= 1. \end{aligned}$$

Once again, how accurate is your numerical solution?

(e) For each of the three methods, the error at $x = 1$ seems to be significantly larger than elsewhere. Why is that?

2. Consider the function $f(x, y) = x^3 + y^3 - 2x^2 + 3y^2 - 8$ (for real x and y).

(a) Find all critical points of f by sketching in the (x, y) plane the iso-contours $\frac{\partial f}{\partial x} = 0$ and $\frac{\partial f}{\partial y} = 0$. Then classify the critical points into local minima, local maxima, and saddle points by considering nearby gradient directions.

(b) Show how steepest descent would behave starting from the point $(x, y) = (1, -1)$. Use the version of steepest descent that moves to the nearest local minimum on the negative gradient line. How many such steepest descent steps are needed to converge to an overall local minimum of f?

3. Let Q be a real symmetric positive definite $n \times n$ matrix.

   (a) Show that any two eigenvectors of Q corresponding to *distinct* eigenvalues of Q are Q-orthogonal. Show this directly from the definition of eigenvector.

   (b) Now enhance the argument from part (a) to establish Q-orthogonality more generally, as follows:

   As we noted earlier in the course, one can find a basis of eigenvectors of Q that are pairwise orthogonal (in the usual sense), even if Q has repeated eigenvalues. Using this fact, show that *any* two such basis vectors are in fact also Q-orthogonal.

4. (a) Show that in the purely quadratic form of the conjugate gradient method, $d_k^T Q d_k = -d_k^T Q g_k$. Using this show that to obtain x_{k+1} from x_k it is necessary to use Q only to evaluate g_k and Qg_k.

   (See the notes for notation:

   - "Purely quadratic" means the conjugate gradient method applied to functions of the form $f(x) = c + b^T x + \frac{1}{2} x^T Q x$, where Q is a real symmetric positive definite $n \times n$ matrix.
   - g_k is the gradient of $f(x)$ evaluated at x_k, and d_k is the descent direction leading from x_k to x_{k+1}.)

   (b) Show that in the purely quadratic form Qg_k can be evaluated by taking a unit step from x_k in the direction of the negative gradient and then evaluating the gradient there. Specifically, if $y_k = x_k - g_k$ and $p_k = \nabla f(y_k)$, then $Qg_k = g_k - p_k$.

   (c) Combine the results of parts (a) and (b) to derive a conjugate gradient method for general functions f much in the spirit of the algorithm presented in class, but which does not require knowledge of the Hessian of f or a line search. (Recall that a line search is used to find the minimum of f along a particular direction.)

5. Find the rectangle of a given perimeter that has the greatest area, by solving the Lagrange multiplier first-order necessary conditions. Verify the second-order sufficiency conditions.

6. In this problem, you are going to use trajectory optimization to find obstacle-free paths for a robot. You will start with an obstacle cost world and a straight line path that blasts through the obstacles, as in Figures 1 and 2 on page 4. We are giving you some MATLAB code to make it easier, in `trajectory_optimization.m`, where you just have to plug in your optimization code.

   (a) Starting from the straight line path that goes across the obstacle field as in Figure 2, perform gradient descent to find a path with optimal cost. You should represent the path as a sequence of points, view those points in some finite-dimensional space, and take gradient steps in that space. Rather than optimize along the entire gradient line as we did in lecture, simply scale the negative gradient by 0.1 and take a corresponding step.

   Remember, this is a path for your robot, so make sure you don't accidentally optimize its start and goal as well; those should remain fixed.

   Plot the path after one iteration — notice that it is moving away from the obstacles. Plot the path after convergence. What happened to it?

(b) To avoid the issue from part (a), you need to tell the optimizer that this is a path instead of some independent points.

One possibility is to consider an additional cost term, $\min\left(\frac{1}{2}\left|\left|\xi_i - \xi_{i-1}\right|\right|^2\right)$, that tells each point to be close to the previous point on the path. This additional term is sometimes called a *smoothness* cost.

Augment your negative gradient from part (a) with a new negative gradient, obtained for each ξ_i from the single term $\frac{1}{2}\left|\left|\xi_i - \xi_{i-1}\right|\right|^2$. Weight the negative obstacle gradient from part (a) with a 0.8 and the new negative gradient term with a 4, add them together, then again scale that vector by 0.1 and take a corresponding step.

(Weights like this are never set in stone; if you wish, you should feel free to experiment with other values.)

Plot the path after 100, 200 and 500 iterations. Why doesn't this work?

(c) Finally, try augmenting the obstacle gradient with a different smoothness cost, telling each point to be close to both its previous and next point:

$$\min\left(\frac{1}{2}\left|\left|\xi_i - \xi_{i-1}\right|\right|^2 + \frac{1}{2}\left|\left|\xi_{i+1} - \xi_i\right|\right|^2\right).$$

Taking the gradient with respect to ξ_i yields $(\xi_i - \xi_{i-1}) + (\xi_i - \xi_{i+1})$, which is $-\xi_{i-1} + 2\xi_i - \xi_{i+1}$.

Perform another optimization using this new smoothness cost to compute gradient updates:

i. Use 0.8 to weight the obstacle cost and 4 to weight the smoothness cost. Then use a step-size scaling of 0.1 as before.

ii. Again, you should feel free to use different weights if your code works better with those, so long as the weights are not too different from those we suggested.

Plot the path after 100 and 5000 iterations. Why is this version better?

(d) What will happen if you were to start the procedure from different initial trajectories? Will the final answer be the same every time? Why?

(e) Suppose that the final solution trajectory that gradient descent comes up with for some initial trajectory for our original cost function is not feasible for the robot to execute (it still passes through some high cost regions that you would rather not have the robot pass through)? What simple common-sense strategy can you come up with to try to mitigate this problem? (*No need to write code unless necessary. We expect a very short description of the strategy only.*)

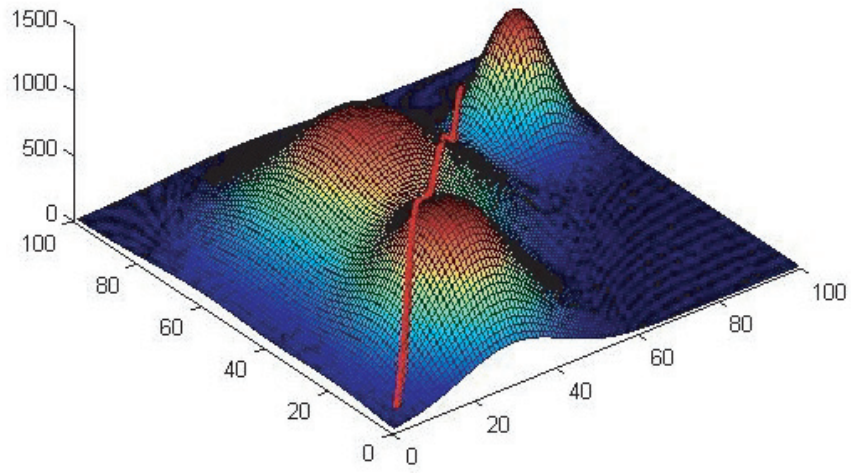

Figure 1: A 2D world with obstacles depicted as cost functions. Shown in red is a straight line path between two locations. The path has high cost because it passes through/near obstacles.

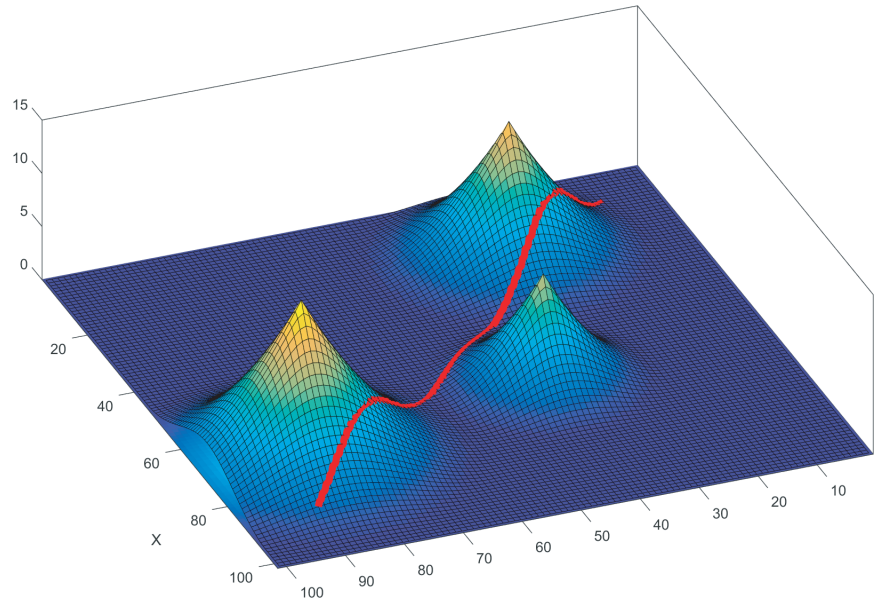

Figure 2: Another 2D obstacle world and straight line path with moderately-high cost, corresponding to the code appearing in `trajectory_optimization.m`.