Assignment 4

Math Fundamentals for Robotics 16-811, Fall 2018
DUE: Thursday, October 25, 2018

[redacted](Andrew ID: [redacted])

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

$$\frac{dy}{dx} = \frac{1}{3y^2} \quad \textit{with } y(2) = 1$$

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

SOLUTION:

$$3y^2 dy - dx = 0$$
$$\rightarrow \quad y^3 - x = C$$

Since $y(2) = 1$, we have $C = -1$. So

$$y = (x-1)^{\frac{1}{3}}$$

(b) Implement and use Eulers 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 Eulers 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.)

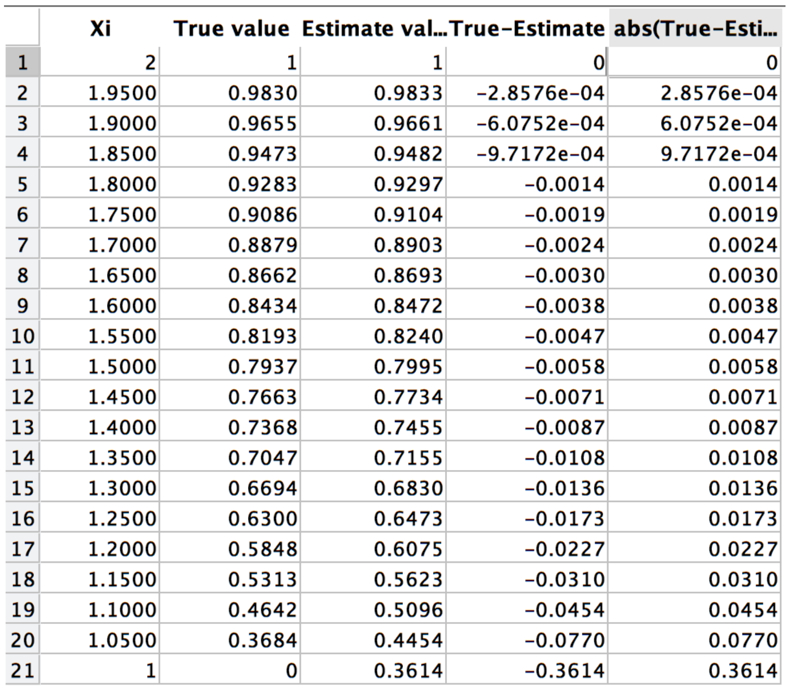
SOLUTION:

For this part, the code is implemented in `code/q1.m`

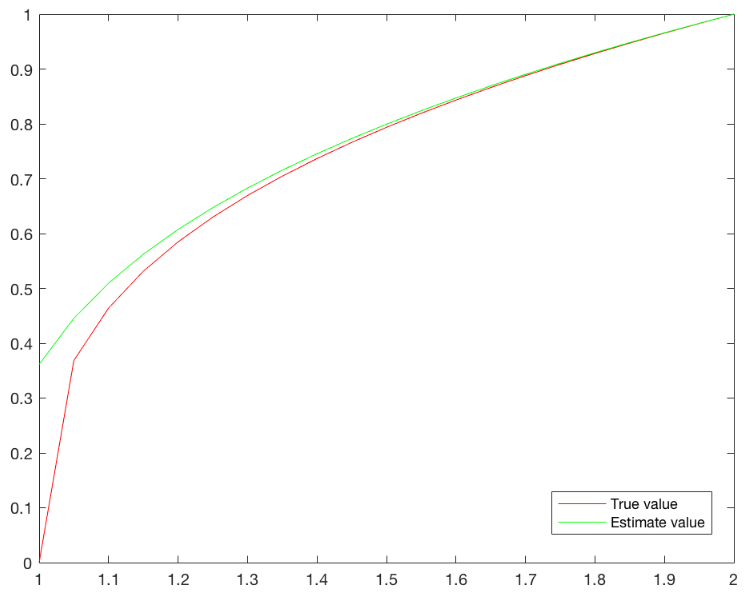
Function `estimate_y = euler(x, x0, y0, h)` is implemented to solve the differential equation numerically using Eulers method. The inputs include:
- `x`: A matrix with size $N \times 1$, containing all the values of x_i.
- `x0`: A value of x_i with known y_i.
- `y0`: The corresponding value of $y(x_i)$.
- `h`: The step size.

The function would be called in function `main()`. In function `main()`, it then would plot the solutions in a table and in a figure and then return the average error.

The table and graph showing the true values and estimated values are shown as below. The returned average error is **0.0295**.



	Xi	True value	Estimate val...	True-Estimate	abs(True-Esti...
1	2	1	1	0	0
2	1.9500	0.9830	0.9833	-2.8576e-04	2.8576e-04
3	1.9000	0.9655	0.9661	-6.0752e-04	6.0752e-04
4	1.8500	0.9473	0.9482	-9.7172e-04	9.7172e-04
5	1.8000	0.9283	0.9297	-0.0014	0.0014
6	1.7500	0.9086	0.9104	-0.0019	0.0019
7	1.7000	0.8879	0.8903	-0.0024	0.0024
8	1.6500	0.8662	0.8693	-0.0030	0.0030
9	1.6000	0.8434	0.8472	-0.0038	0.0038
10	1.5500	0.8193	0.8240	-0.0047	0.0047
11	1.5000	0.7937	0.7995	-0.0058	0.0058
12	1.4500	0.7663	0.7734	-0.0071	0.0071
13	1.4000	0.7368	0.7455	-0.0087	0.0087
14	1.3500	0.7047	0.7155	-0.0108	0.0108
15	1.3000	0.6694	0.6830	-0.0136	0.0136
16	1.2500	0.6300	0.6473	-0.0173	0.0173
17	1.2000	0.5848	0.6075	-0.0227	0.0227
18	1.1500	0.5313	0.5623	-0.0310	0.0310
19	1.1000	0.4642	0.5096	-0.0454	0.0454
20	1.0500	0.3684	0.4454	-0.0770	0.0770
21	1	0	0.3614	-0.3614	0.3614



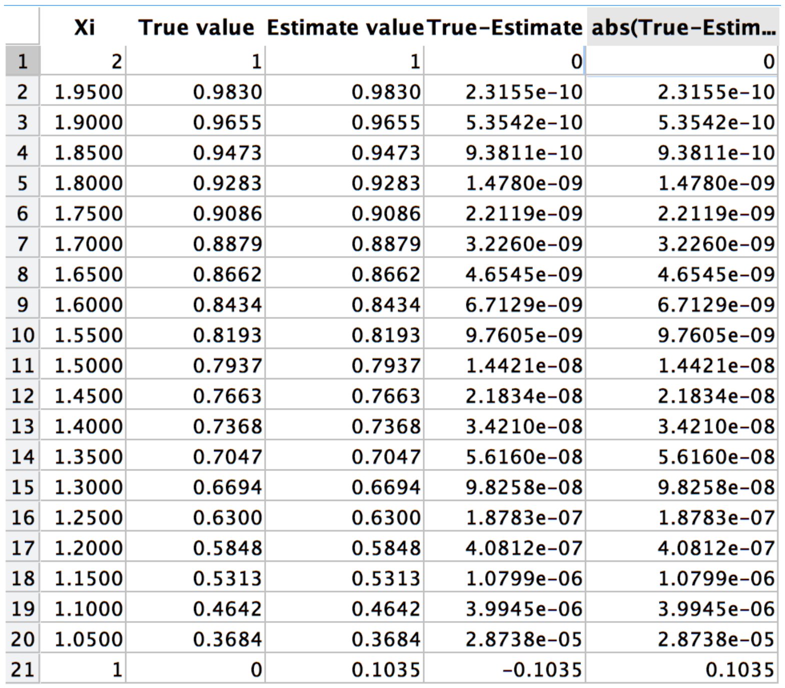
(c) Implement and use a fourth-order Runge-Kutta method to solve the differential equation numerically. Again, use a step size of 0.05. Again, how accurate is your numerical solution?

SOLUTION:

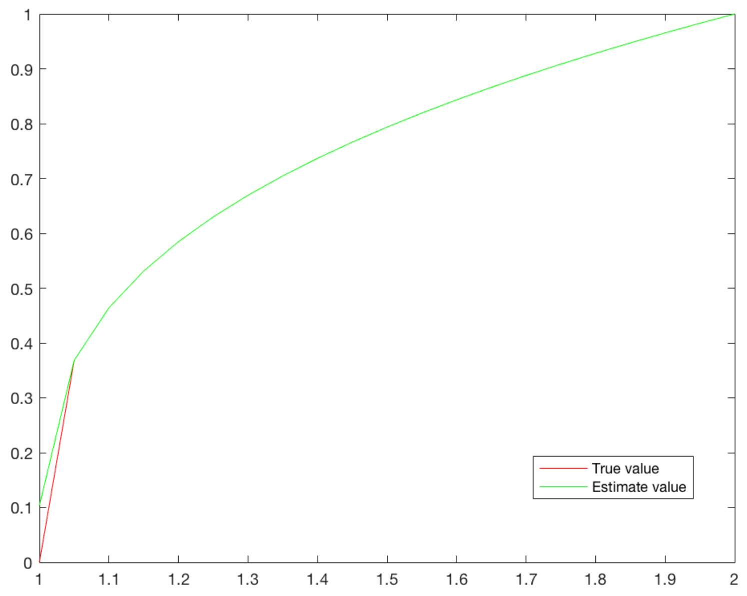
For this part, the code is implemented in `code/q1.m`

Function `estimate_y = RK4(x, x0, y0, h)` is implemented to solve the differential equation numerically using Runge-Kutta method. The inputs and outputs of the function are just the same as those in function `estimate_y = euler(x, x0, y0, h)` in part (b).

The table and graph showing the true values and estimated values are shown as below. The returned error is **0.0049**.



	Xi	True value	Estimate value	True-Estimate	abs(True-Estim…
1	2	1	1	0	0
2	1.9500	0.9830	0.9830	2.3155e-10	2.3155e-10
3	1.9000	0.9655	0.9655	5.3542e-10	5.3542e-10
4	1.8500	0.9473	0.9473	9.3811e-10	9.3811e-10
5	1.8000	0.9283	0.9283	1.4780e-09	1.4780e-09
6	1.7500	0.9086	0.9086	2.2119e-09	2.2119e-09
7	1.7000	0.8879	0.8879	3.2260e-09	3.2260e-09
8	1.6500	0.8662	0.8662	4.6545e-09	4.6545e-09
9	1.6000	0.8434	0.8434	6.7129e-09	6.7129e-09
10	1.5500	0.8193	0.8193	9.7605e-09	9.7605e-09
11	1.5000	0.7937	0.7937	1.4421e-08	1.4421e-08
12	1.4500	0.7663	0.7663	2.1834e-08	2.1834e-08
13	1.4000	0.7368	0.7368	3.4210e-08	3.4210e-08
14	1.3500	0.7047	0.7047	5.6160e-08	5.6160e-08
15	1.3000	0.6694	0.6694	9.8258e-08	9.8258e-08
16	1.2500	0.6300	0.6300	1.8783e-07	1.8783e-07
17	1.2000	0.5848	0.5848	4.0812e-07	4.0812e-07
18	1.1500	0.5313	0.5313	1.0799e-06	1.0799e-06
19	1.1000	0.4642	0.4642	3.9945e-06	3.9945e-06
20	1.0500	0.3684	0.3684	2.8738e-05	2.8738e-05
21	1	0	0.1035	-0.1035	0.1035



(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:

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

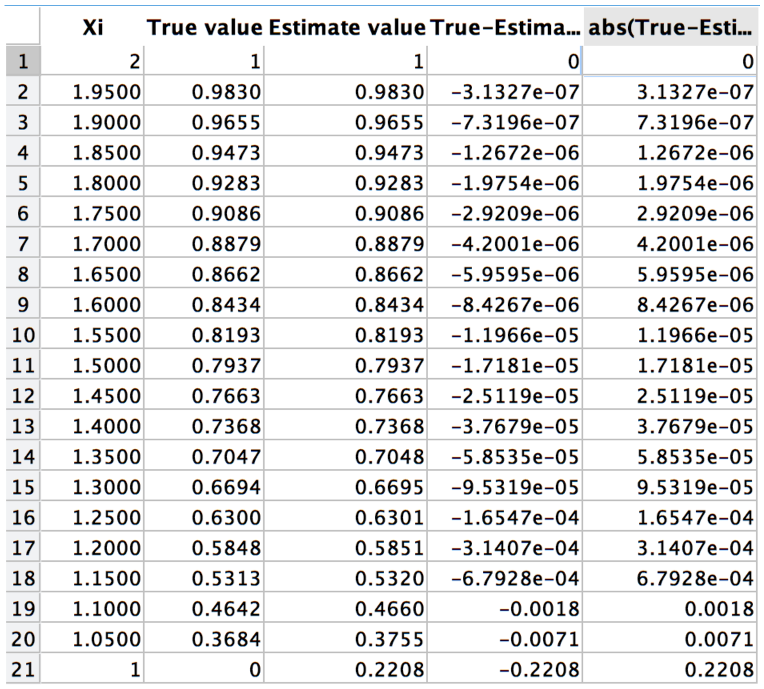
Once again, how accurate is your numerical solution?

SOLUTION:

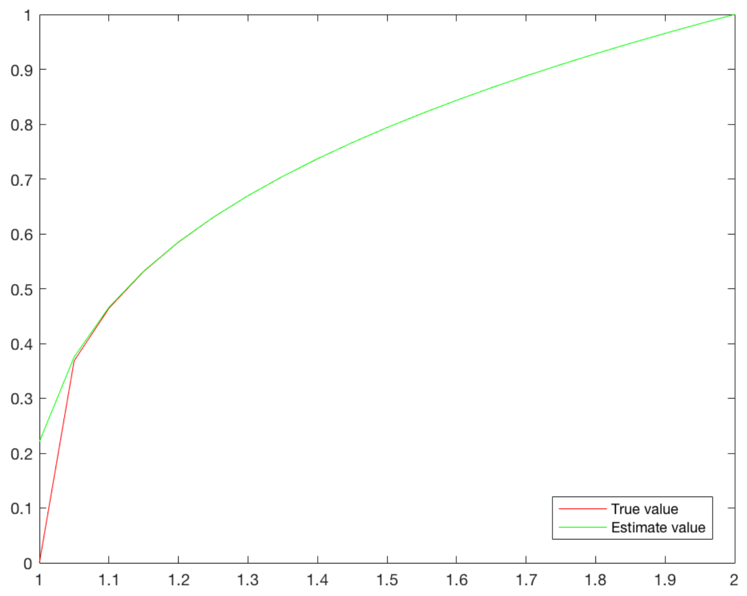
For this part, the code is implemented in `code/q1.m`

Function `estimate_y = AB4(x, x0, y0, h)` is implemented to solve the differential equation numerically using Adams-Bashforth method. The inputs and outputs of the function are just the same as those in function `estimate_y = euler(x, x0, y0, h)` in part (b).

The table and graph showing the true values and estimated values are shown as below. The returned average error is **0.0110**.



	Xi	True value	Estimate value	True−Estima…	abs(True−Esti…
1	2	1	1	0	0
2	1.9500	0.9830	0.9830	−3.1327e−07	3.1327e−07
3	1.9000	0.9655	0.9655	−7.3196e−07	7.3196e−07
4	1.8500	0.9473	0.9473	−1.2672e−06	1.2672e−06
5	1.8000	0.9283	0.9283	−1.9754e−06	1.9754e−06
6	1.7500	0.9086	0.9086	−2.9209e−06	2.9209e−06
7	1.7000	0.8879	0.8879	−4.2001e−06	4.2001e−06
8	1.6500	0.8662	0.8662	−5.9595e−06	5.9595e−06
9	1.6000	0.8434	0.8434	−8.4267e−06	8.4267e−06
10	1.5500	0.8193	0.8193	−1.1966e−05	1.1966e−05
11	1.5000	0.7937	0.7937	−1.7181e−05	1.7181e−05
12	1.4500	0.7663	0.7663	−2.5119e−05	2.5119e−05
13	1.4000	0.7368	0.7368	−3.7679e−05	3.7679e−05
14	1.3500	0.7047	0.7048	−5.8535e−05	5.8535e−05
15	1.3000	0.6694	0.6695	−9.5319e−05	9.5319e−05
16	1.2500	0.6300	0.6301	−1.6547e−04	1.6547e−04
17	1.2000	0.5848	0.5851	−3.1407e−04	3.1407e−04
18	1.1500	0.5313	0.5320	−6.7928e−04	6.7928e−04
19	1.1000	0.4642	0.4660	−0.0018	0.0018
20	1.0500	0.3684	0.3755	−0.0071	0.0071
21	1	0	0.2208	−0.2208	0.2208



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

SOLUTION:

The equation $y(x) = (x-1)^{\frac{1}{3}}$ is not differentiable at $x = 1$

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.

SOLUTION:

The code for sketching the iso-contours is implemented in `code/q2a.m`

$$\frac{\partial f}{\partial x} = 3x^2 - 4x = 0$$

$$\rightarrow x_1 = 0, \quad x_2 = \frac{4}{3}$$

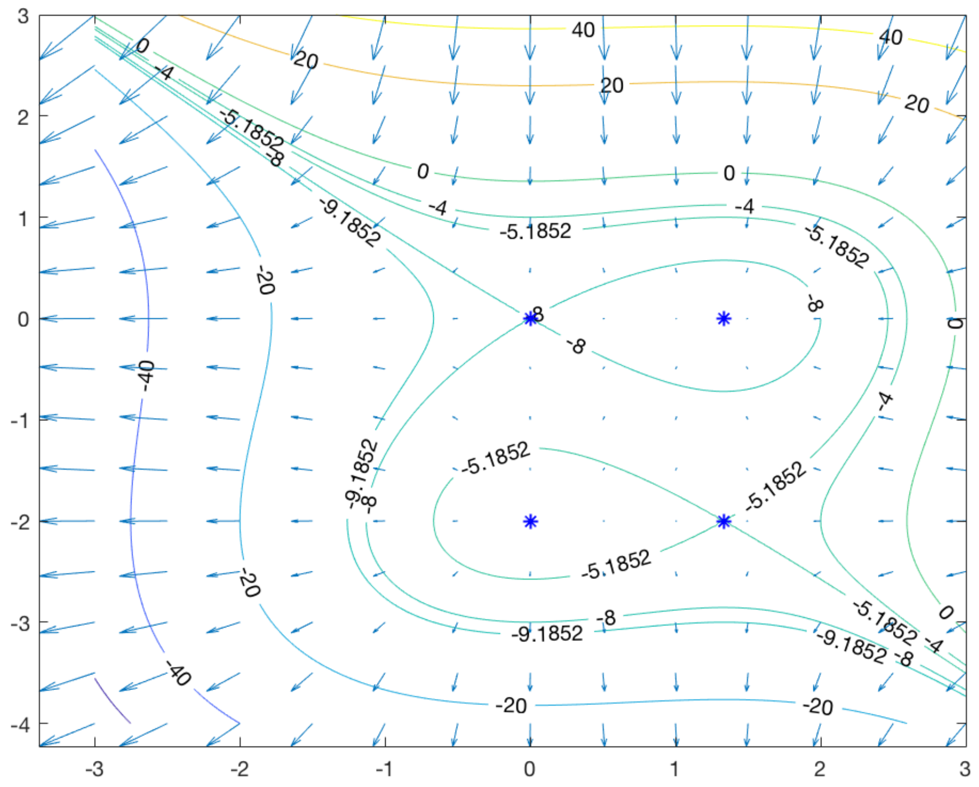
$$\frac{\partial f}{\partial y} = 3y^2 + 6y = 0$$

$$\rightarrow y_1 = 0, \quad y_2 = -2$$

The critical points of f include:

$$(0, 0), \quad (0, -2), \quad (\frac{4}{3}, 0), \quad (\frac{4}{3}, -2)$$

The plotted iso-contours in the (x, y) plane and the gradients are shown as below. The critical points are also plotted in this figure and the gradients shown in the figure is in the gradient descent direction. So we can classify the critical points



into local minima, local maxima and saddle points by considering the nearby gradient directions:

• (0, -2) is local maxima.

• $(\frac{4}{3}, 0)$ is local minima.

• (0, 0) and $(\frac{4}{3}, -2)$ are saddle points.

(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?

SOLUTION:

This part is implemented in `code/q2b.m`

$$\nabla f = \begin{bmatrix} 3x^2 - 4x \\ 3y^2 + 6y \end{bmatrix}$$

• step 1:

$$\mathbf{x}^{(0)} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \text{ so } \mathbf{u}^{(0)} = \nabla f(\mathbf{x}^{(0)}) = \begin{bmatrix} -1 \\ -3 \end{bmatrix}$$

$$\begin{aligned} g(t) &= f(\mathbf{x}^{(0)} - t\mathbf{u}^{(0)}) \\ &= (1+t)^3 + (-1+3t)^3 - 2(1+t)^2 + 3(-1+3t)^2 - 8 \end{aligned}$$

$\frac{\partial g(t)}{\partial t} = 3(1+t)^2 + 9(-1+3t)^2 - 4(1+t) + 18(-1+3t) = 0$

$\rightarrow t_1 = -\frac{5}{14}, \quad t_2 = \frac{1}{3}$

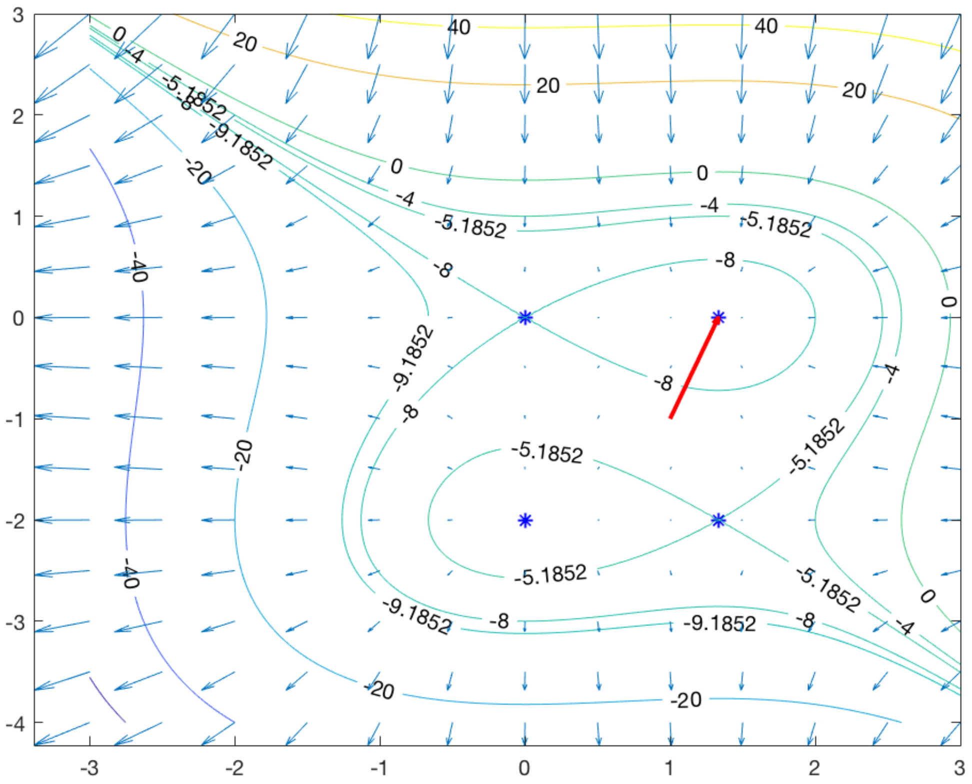
When $t^* = \frac{1}{3}$, it can minimize $g(t)$.

$$\mathbf{x}^{(1)} = \mathbf{x}^{(0)} - t^*\mathbf{u}^{(0)} = \begin{bmatrix} 4/3 \\ 0 \end{bmatrix}$$

$$\mathbf{u}^{(1)} = \nabla f(\mathbf{x}^{(1)}) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

So one such steepest descent step is needed to converge to an overall local minimum of f.

The path of moving to the nearest local minimum on the negative gradient line is show as the red path in the below figure.



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.

   SOLUTION:

   Suppose $\mathbf{v}_i$ and $\mathbf{v}_j$ are two eigenvectors of Q corresponding to two distinct eigenvalues λ_i and λ_j. Then we have $Q\mathbf{v}_i = \lambda_i \mathbf{v}_i$ and $Q\mathbf{v}_j = \lambda_j \mathbf{v}_j$.
   Thus, we have $\mathbf{v}_j^T Q\mathbf{v}_i = \lambda_i \mathbf{v}_j^T \mathbf{v}_i$ and $\mathbf{v}_i^T Q\mathbf{v}_j = \lambda_j \mathbf{v}_i^T \mathbf{v}_j$.
   So, $(\mathbf{v}_j^T Q\mathbf{v}_i)^T = \mathbf{v}_i^T Q^T \mathbf{v}_j = \lambda_i \mathbf{v}_j^T \mathbf{v}_i$.
   Since Q is symmetric, $Q^T = Q$, we now have $\lambda_i \mathbf{v}_j^T \mathbf{v}_i = \lambda_j \mathbf{v}_i^T \mathbf{v}_j$, namely, $(\lambda_i - \lambda_j)\mathbf{v}_j^T \mathbf{v}_i = 0$. And we know that $\lambda_i \neq \lambda_j$, so $\mathbf{v}_j^T \mathbf{v}_i = 0$. That is, the two eigenvectors $\mathbf{v}_i$ and $\mathbf{v}_j$ are orthogonal, so we have

   $$\mathbf{v}_j^T Q\mathbf{v}_i = \lambda_i \mathbf{v}_j^T \mathbf{v}_i = 0$$

   Thus, any two eigenvectors of Q corresponding to distinct eigenvalues of Q are Q-orthogonal.

   (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.

   SOLUTION:

   Suppose $\mathbf{v}_i$ and $\mathbf{v}_j$ are two eigenvectors of Q. So there is $<\mathbf{v}_i, \mathbf{v}_j>= \mathbf{v}_j^T \mathbf{v}_i = 0$. Then we have $Q\mathbf{v}_i = \lambda_i \mathbf{v}_i$. Since the two eigenvectors $\mathbf{v}_i$ and $\mathbf{v}_j$ are orthogonal, we have

   $$\mathbf{v}_j^T Q\mathbf{v}_i = \lambda_i \mathbf{v}_j^T \mathbf{v}_i = 0$$

   Thus, any two orthogonal eigenvectors of Q are 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.

   SOLUTION:

   In the conjugate gradient method, we have:

   $$\beta_k = \frac{g_{k+1}^T Q d_k}{d_k^T Q d_k}$$

   $$d_{k+1} = -g_{k+1} + \beta_k d_k$$

So there is:

$$\begin{aligned} d_k &= -g_k + \beta_{k-1} d_{k-1} \\ &= -g_k + \frac{g_k^T Q d_{k-1}}{d_{k-1}^T Q d_{k-1}} \end{aligned}$$

So,

$$\begin{aligned} d_k^T Q d_k &= d_k^T Q(-g_k + \beta_{k-1} d_{k-1}) \\ &= -d_k^T Q g_k + \beta_{k-1} d_k^T Q d_{k-1} \\ &= -d_k^T Q g_k \end{aligned}$$

So $d_k^T Q d_k = -d_k^T Q g_k$ is proved.

$$\begin{aligned} x_{k+1} &= x_k + \alpha_k d_k \\ &= x_k - \frac{d_k^T g_k}{d_k^T Q d_k} d_k \\ &= x_k + \frac{d_k^T g_k}{d_k^T Q g_k} d_k \end{aligned}$$

$g_k = Qx_k + b$

So, in order to obtain x_{k+1} from x_k, it is necessary to use Q only to evaluate g_k and Qg_k.

(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 $p_k = \Delta f(y_k)$, then $Qg_k = g_k - p_k$.

SOLUTION:

$$y_k = x_k - g_k$$
$$p_k = \nabla f(y_k) = Qy_k + b$$

$$\begin{aligned} \rightarrow \quad Qg_k &= Q(x_k - y_k) \\ &= Qx_k - Qy_k \\ &= Qx_k - p_k - b \\ &= (Qx_k - b) - p_k \\ &= g_k - p_k \end{aligned}$$

(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.)

SOLUTION:

According to the results of parts (a) and (b), we have:

$$\begin{aligned} x_{k+1} &= x_k + \frac{d_k^T g_k}{d_k^T Q g_k} d_k \\ &= x_k + \frac{d_k^T g_k}{d_k^T (g_k - p_k)} d_k \\ &= x_k + \frac{d_k^T g_k}{d_k^T g_k - d_k^T p_k} d_k \end{aligned}$$

Since we know that:

$$\begin{aligned} d_k^T g_k &= d_k^T (Q(x_0 + \alpha_0 d_0 + \cdots + \alpha_{k-1} d_{k-1}) + b) \\ &= d_k^T Q x_0 + \alpha_0 d_k^T Q d_0 + \cdots + \alpha_{k-1} d_k^T Q d_{k-1} + d_k^T b \\ &= d_k^T (Q x_0 + b) \\ &= d_k^T g_0 \end{aligned}$$

So,

$$x_{k+1} = x_k + \frac{d_k^T g_0}{d_k^T g_0 - d_k^T p_k} d_k$$

This does not require knowledge of the Hessian of f or a line search.

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.

SOLUTION:

Suppose the given perimeter is denoted as c. Then the problem would be to solve:

$$\max f(x,y) = xy \quad s.t. \quad h(x,y) = 2x + 2y - c = 0$$

and this is equal to solve:

$$\min f(x,y) = -xy \quad s.t. \quad h(x,y) = 2x + 2y - c = 0$$

So we have

$$\begin{aligned} F(x,y,\lambda) &= f(x,y) + \lambda h(x,y) \\ &= -xy + \lambda(2x + 2y - c) \end{aligned}$$

$$\frac{\partial F}{\partial x} = -y + 2\lambda$$
$$\frac{\partial F}{\partial y} = -x + 2\lambda$$
$$\frac{\partial F}{\partial \lambda} = 2x + 2y - c$$

By setting $\nabla F = 0$, we get:

$$x = \frac{c}{2}$$
$$y = \frac{c}{2}$$
$$\lambda = \frac{c}{4}$$

So when $x = y = c/2$, the rectangle has the greatest area.

For second order condition, we have:

$$[\nabla^2 f] = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}$$

$$[\nabla^2 h] = \begin{bmatrix} \frac{\partial^2 h}{\partial x^2} & \frac{\partial^2 h}{\partial x \partial y} \\ \frac{\partial^2 h}{\partial y \partial x} & \frac{\partial^2 h}{\partial y^2} \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

So,

$$[\nabla^2 f] + \lambda[\nabla^2 h] = \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix}$$

For all d s.t. $d^T \nabla h = d^T \begin{bmatrix} 2 \\ 2 \end{bmatrix} = 0$, namely, $d = m \begin{bmatrix} 1 \\ -1 \end{bmatrix}$ where m is an arbitrary real number. Then we have:

$$d^T([\nabla^2 f] + \lambda[\nabla^2 h])d = [m \;\; -m] \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} \begin{bmatrix} m \\ -m \end{bmatrix} = 2m^2 \geq 0$$

So the second-order sufficiency conditions is verified.

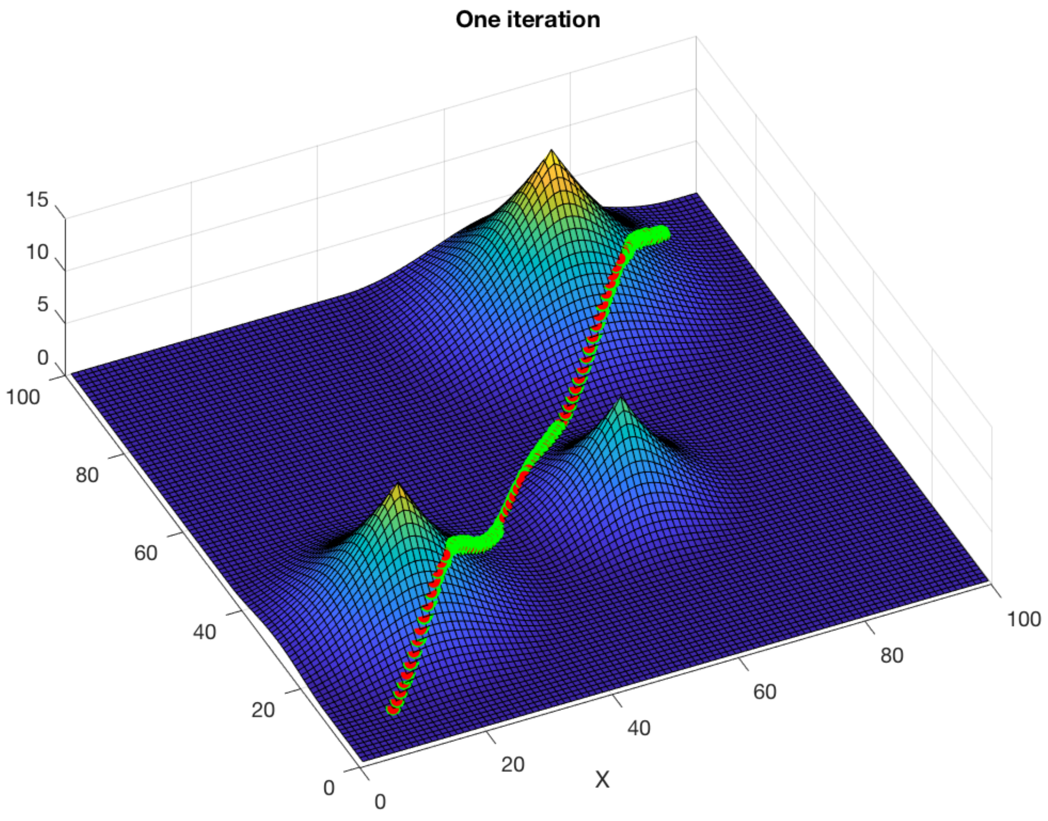
6. In this problem, you are going to use trajectory optimization to find obstacle-free paths for a robot. You will start with an obstacle cost world and a straight line path that blasts through the obstacles, as in Figures 1 and 2 on page 4. We are giving you some MATLAB code to make it easier, in `trajectory_optimization.m`,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

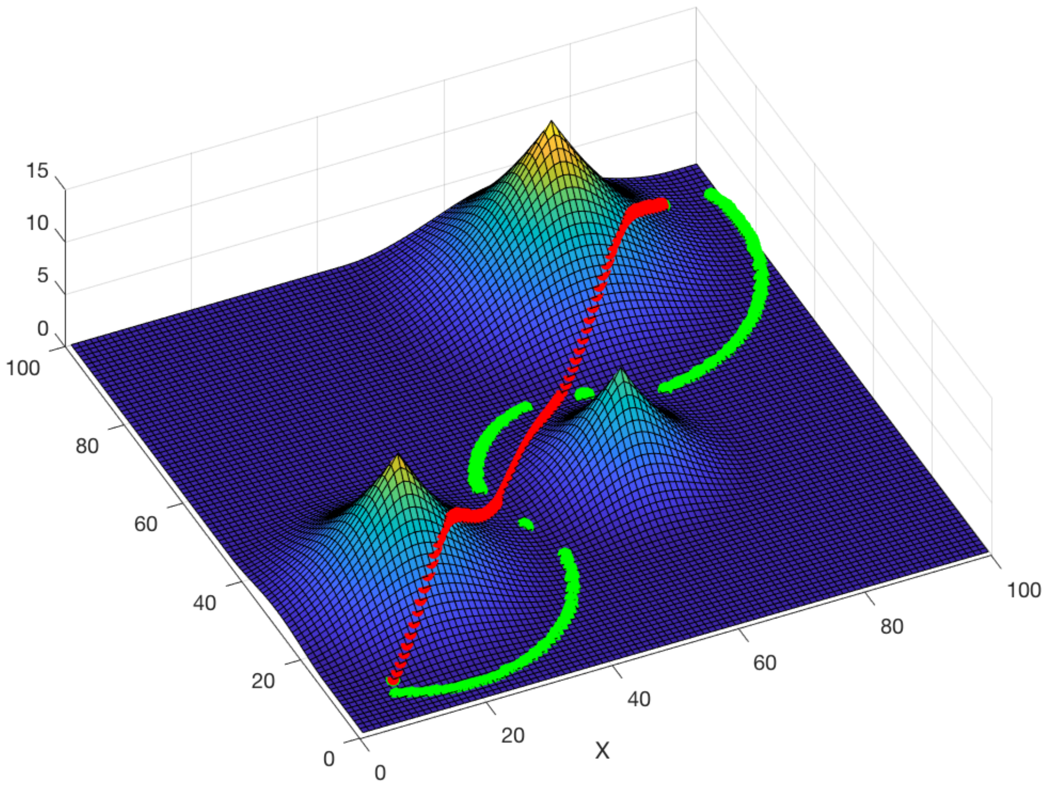
   SOLUTION:

   This part is implemented in `code/q6a.m`.

   The path after one iteration of gradient descent is shown as below:



After convergence, the path is shown as below:



Since the path is a sequence of independent points, after converge on gradient descent, the points are isolated and they can not generate a completed path.

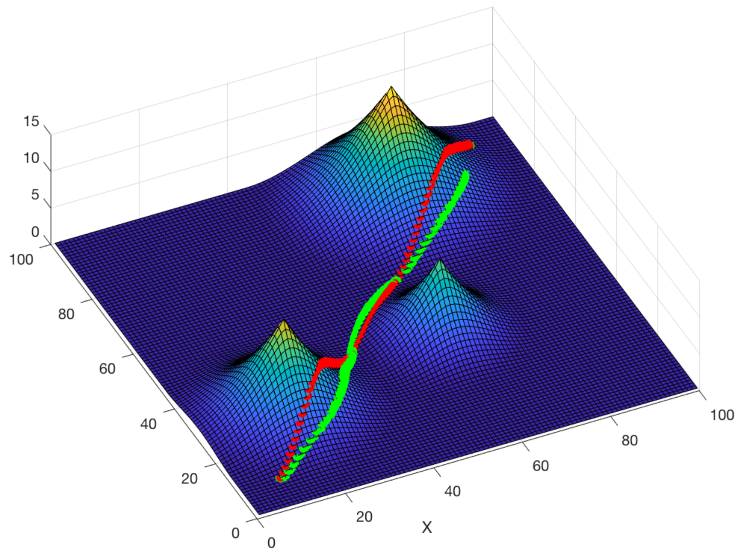
(b) To avoid the issue from part (a), you need to tell the optimizer that this is a path instead of some independent points.
One possibility is to consider an additional cost term, $\min(\frac{1}{2}||\xi_i - \xi_{i-1}||^2)$, that tells each point to be close to the previous point on the path. This additional 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 $\frac{1}{2}||\xi_i - \xi_{i-1}||^2$. 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?

SOLUTION:

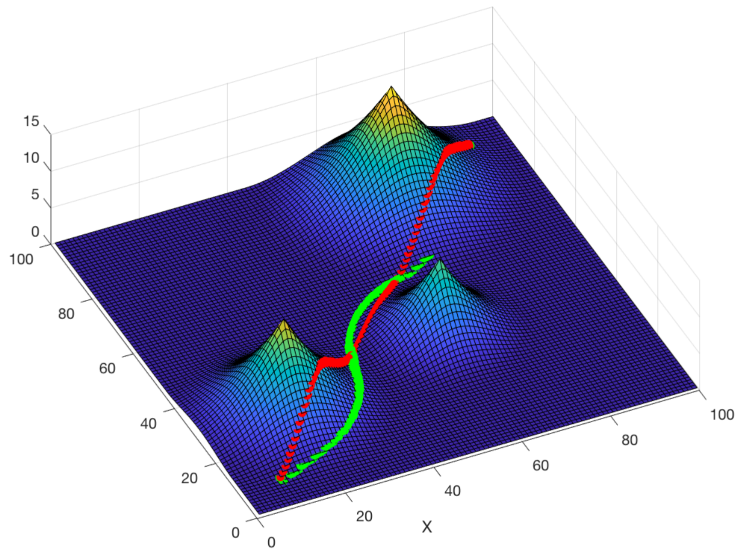
This part is implemented in `code/q2b.m`

The gradient respect to ξ_i of the additional cost term is $(\xi_i - \xi_{i-1})$.

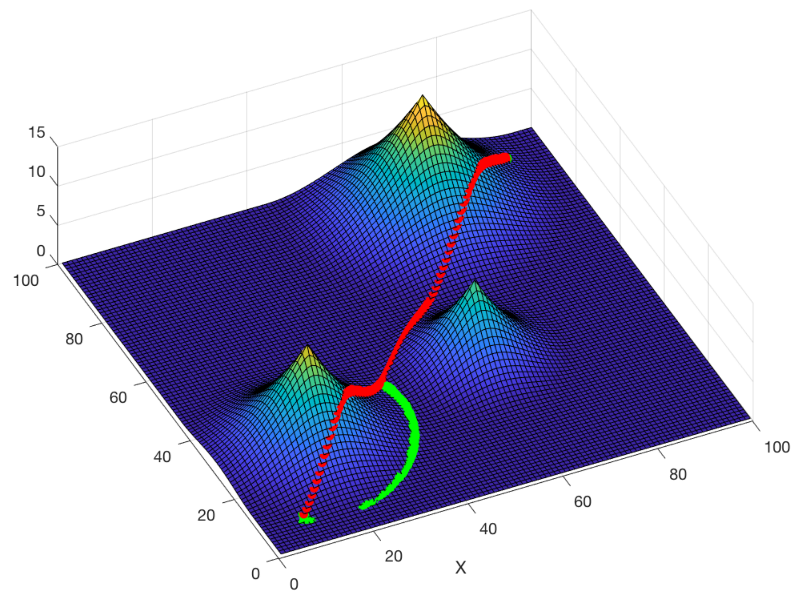
After adding the new negative gradient term into the function, the path after 100 iterations is shown as below:



The path after 200 iterations is shown as below:



The path after 500 iterations is shown as below:



The reason that this doesn't work is that, all the points are optimized to be close to the precious point on the path, so these points tend to 'gather' forward. So it can not form a full path.

(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:

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

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.

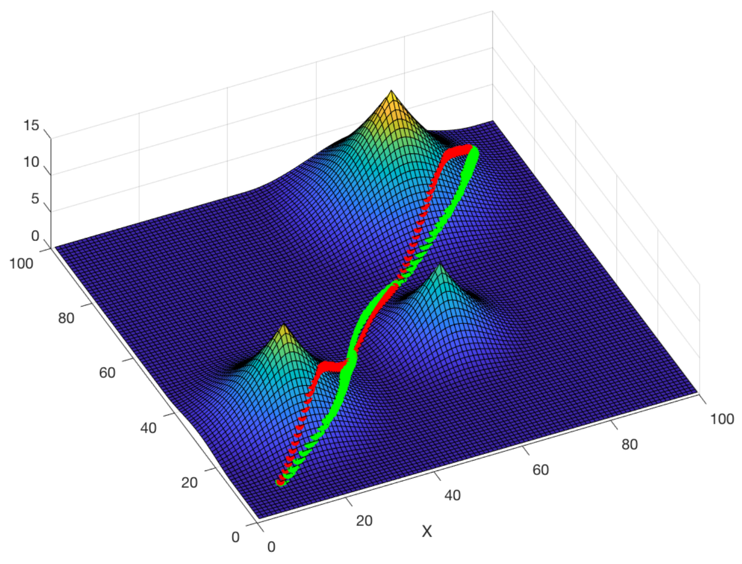
ii. Again, you should feel free to use different weights if your code works better with those, so long as the weights are not too different from those we suggested.

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

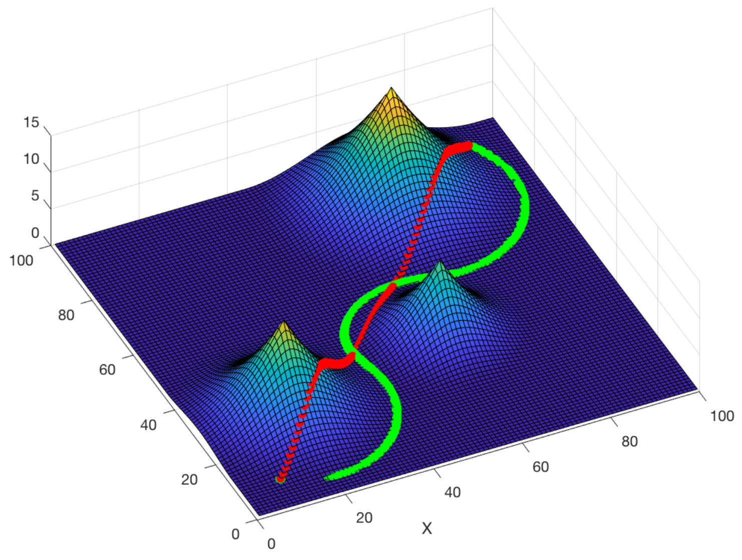
SOLUTION:

This part is implemented in `code/q6c.m`

After adding the new negative gradient term into the function, the path after 100 iterations is shown as below:



The path after 5000 iterations is shown as below:



The reason that this version is better is because that it also take both the previous point and the following point into consideration. Each point is optimized to be close to its previous point and its following point. So each two adjacent points on the path are close to each other and can form a full path.

(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?

SOLUTION:

With different initial trajectories, the optimized path might be different and the final answer would not be the same every time. This is because that the points at different locations have different gradients values and directions. So with different initial path, the optimization process might guide the points into different directions. Thus the final result would not always be the same.

(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.)

SOLUTION:

For points that have high cost, we can perform gradient descent without considering the additional smoothness cost to optimize them. This might cause isolated points. Then we can connect the isolated point with its previous point and its following point on the path by adding points between them with linear interpolation. For those added points, we can further optimize them with gradient descent method. Keep optimizing and adding points, finally we can get a full path.