1. (a)

I did this homework with Luning Zhao.

We work on our homework seperately, but we discuss when meeting problems.

Homework is fine.

(b) I certify that all solutions are entirely in my words. and that I have not looked at another student's solutions. I have credited all external sources in this write up.

Siyao Jia

2. (a) For $\forall \lambda \in [0,1]$, $\lambda f(x_0) + (1-\lambda) f(x_1)$

$= \lambda \|x_0 - b\|_2 + (1-\lambda) \|x_1 - b\|_2$

$f(\lambda x_0 + (1-\lambda) x_1) = \|\lambda x_0 + (1-\lambda) x_1 - b\|_2$

$= \|\lambda (x_0 - b) + (1-\lambda)(x_1 - b)\|_2$

$\leq \|\lambda (x_0 - b)\|_2 + \|(1-\lambda)(x_1 - b)\|_2$

$= \lambda \|x_0 - b\|_2 + (1-\lambda) \|x_1 - b\|_2$

$= \lambda f(x_0) + (1-\lambda) f(x_1)$

So $f(x) = \|x - b\|_2$ is a convex function of x.

(b) No, gradient descent will not find the optimal solution

Because the positions output is: [0,0], [0.6, 0.8], [1.2, 1.6], [1.8, 2.4], [2.4, 3.2], [3.4], [3.6, 4.8], [4.2, 5.6], [4.8, 6.4], [4.2, 5.6], [4.8, 6.4]

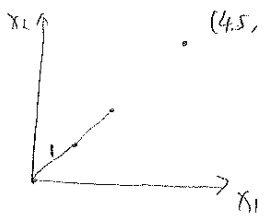
We can see that because $t_i = 1$, the predicted position will stay between [4.2, 5.6] or [4.8, 6.4] so it will never find the optimal solution.

To prove that, first calculate $\nabla f(x_i)$

$f(x) = \|x - b\| = \sqrt{(x_1 - b_1)^2 + (x_2 - b_2)^2}$

$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \end{bmatrix} = \begin{bmatrix} \frac{(x_1 - b_1)}{\sqrt{(x_1 - b_1)^2 + (x_2 - b_2)^2}} \\ \frac{(x_2 - b_2)}{\sqrt{(x_1 - b_1)^2 + (x_2 - b_2)^2}} \end{bmatrix} = \frac{\vec{x}_i - \vec{b}}{\|\vec{x}_i - \vec{b}\|_2}$

$x_0 = [0,0] \longrightarrow x_1 = \frac{\vec{b}}{\|\vec{b}\|} \longrightarrow x_2 = \frac{2\vec{b}}{\|\vec{b}\|}$



So, everytime x_i will move toward $\vec{b}$ by a unit vector, since $\|\vec{b}\|_2$ is not a integer, $\vec{x}_i$ will never goes to $\vec{b}$.

For general $\vec{b}$, if $|\|\vec{b}\|_2 - \text{int}(\|\vec{b}\|_2)| \leq 0.01$, then it takes $\text{int}(\|\vec{b}\|_2)$ to get within 0.01 of the optimal solution. Otherwise, gradient descent will not find the optimal solution

(c) Similar as (b), $\vec{x}_{i+1} = \vec{x}_i - (\frac{5}{6})^i \frac{\vec{x}_i - \vec{b}}{\|\vec{x}_i - \vec{b}\|_2}$

The first step will go toward $\vec{b}$ by $t_1 = 1$, the second step go by $t_2 = (\frac{5}{6})^1$

After n+1 steps, the total length it goes is: $L = \sum_{i=0}^{n} (\frac{5}{6})^i$ ①

$\frac{5}{6} \cdot L = \sum_{i=0}^{n} (\frac{5}{6})^{i+1} = \sum_{i=1}^{n+1} (\frac{5}{6})^i$ ②

① - ② $\Rightarrow$ $\frac{1}{6} L = 1 - (\frac{5}{6})^{n+1}$ $\quad L = 5[\frac{6}{5} - (\frac{5}{6})^n]$

$\|b\| = \sqrt{4.5^2 + 6^2} = 7.5$

We want to find n, s.t. $|L - \|b\|| < 0.01$

but we will notice that: $L = 5(\frac{6}{5} - (\frac{5}{6})^n) \leq 6$, so $|L - \|b\|| > 1.5$

So this gradient descent will not find the optimal solution.

For general $b \neq 0$, if $\|b\| \leq 6.01$, then GD will find the optimal solution

If not, GD will not find the optimal solution.

(d) Similar as (c), after n+1 steps, $L = \sum_{i=0}^{n} \frac{1}{i+1} = \sum_{i=1}^{n+1} \frac{1}{i}$

When n is large (which is the case here, because $\|b\| = 7.5$), $L \approx \log n$

To find n, s.t $\log n \approx 7.5 - 0.01$, $n \approx e^{7.5-0.01} \approx 1791$ step

For general b, this GD should always find the optimal solution

(e) see plots in next page.

The step size of $\frac{1}{i+1}$ works for all choices of A and b.

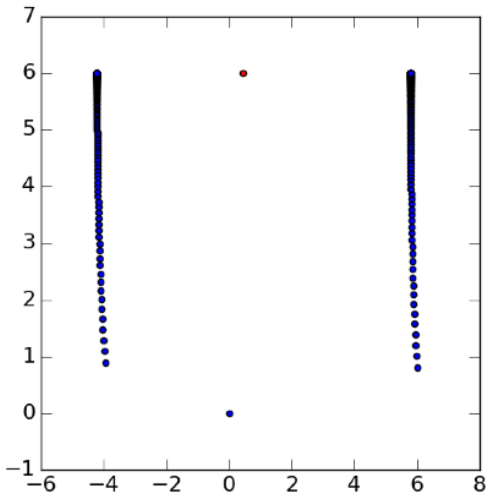
Because step "i" jumps back and forth between two parallel lines, so any points out of those two lines will not be reached.

step "$(\frac{5}{6})^i$" the step size shrinks too fast, although it goes toward the optimal solution, if it's too far away, this step size will be be able to go that far.

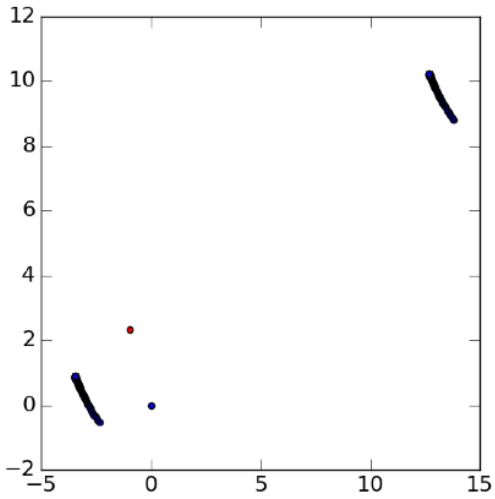
The step "$\frac{1}{i+1}$", although the step size also shrinks, it doesn't have a limit as step "$(\frac{5}{6})^i$", so it will always go to the optimal solution finally.

2 (e)

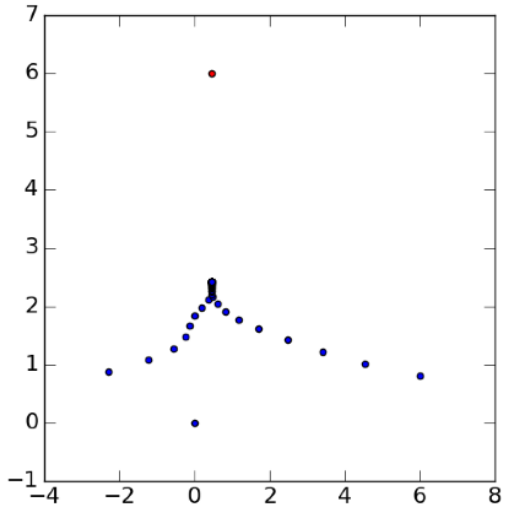
step “1” A[0]



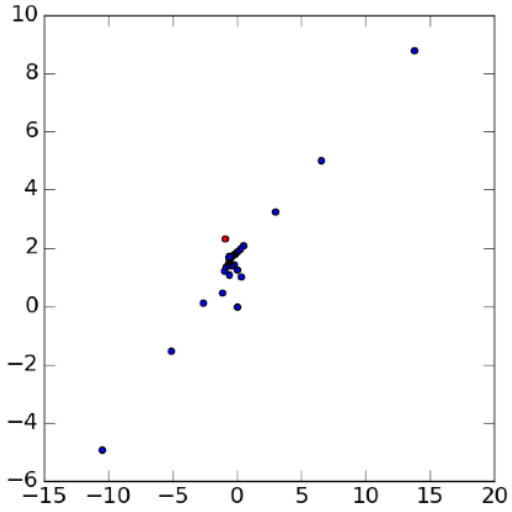
step “1” A[1]



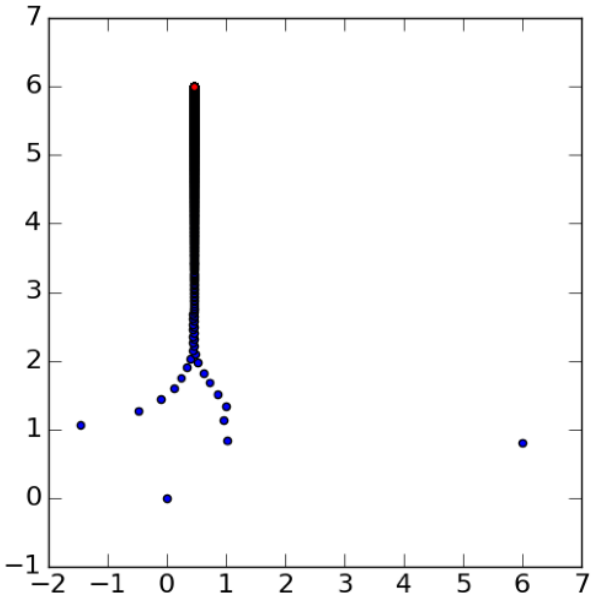
step “(5/6)**i” A[0]



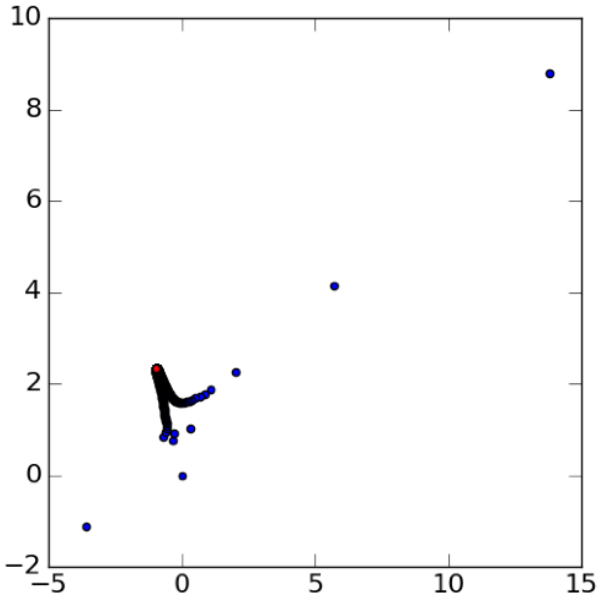
step “(5/6)**i” A[0]



step “1/1+i” A[0]



step “1/1+i” A[0]



```python
""" Tools for calculating Gradient Descent for ||Ax-b||. """
import matplotlib.pyplot as plt
import numpy as np


def main():
    ###########################################################################
    # TODO(student): Input Variables
    #A = np.array([[15, 8], [6, 5]])  # do not change this until the last part
    A = np.array([[10, 0], [0, 1]])  # do not change this until the last part
    b = np.array([4.5, 6])  # b in the equation ||Ax-b||
    initial_position = np.array([0, 0])  # position at iteration 0
    total_step_count = 1000  # number of GD steps to take
    step_size = lambda i: (1./(i + 1.))  # step size at iteration i
    ###########################################################################

    # computes desired number of steps of gradient descent
    positions = compute_updates(A, b, initial_position, total_step_count, step_size)

    # print out the values of the x_i
    print(positions)
    print(np.dot(np.linalg.inv(A), b))

    # plot the values of the x_i
    plt.clf()
    plt.scatter(positions[:, 0], positions[:, 1], c='blue')
    plt.scatter(np.dot(np.linalg.inv(A), b)[0],
                np.dot(np.linalg.inv(A), b)[1], c='red')
    plt.plot()
    plt.show()


def compute_gradient(A, b, x):
    """Computes the gradient of ||Ax-b|| with respect to x."""
    return np.dot(A.T, (np.dot(A, x) - b)) / np.linalg.norm(np.dot(A, x) - b)


def compute_update(A, b, x, step_count, step_size):
    """Computes the new point after the update at x."""
    return x - step_size(step_count) * compute_gradient(A, b, x)


def compute_updates(A, b, p, total_step_count, step_size):
    """Computes several updates towards the minimum of ||Ax-b|| from p.

    Params:
        b: in the equation ||Ax-b||
        p: initialization point
        total_step_count: number of iterations to calculate
        step_size: function for determining the step size at step i
    """
    positions = [np.array(p)]
    for k in range(total_step_count):
        positions.append(compute_update(A, b, positions[-1], k, step_size))
    return np.array(positions)


main()
```

3. (a) $f(\vec{x}) = \frac{1}{2}\|A\vec{x}\|^2 = \frac{1}{2}(A\vec{x})^T(A\vec{x}) = \frac{1}{2}\vec{x}^T A^T A\vec{x}$

$\frac{\partial f(\vec{x})}{\partial \vec{x}} = \vec{x}^T A^T A \qquad \nabla f(\vec{x}) = \left(\frac{\partial f}{\partial \vec{x}}\right)^T = A^T A\vec{x}$

$\vec{x}_{i+1} = \vec{x}_i - \gamma A^T A\vec{x}_i = (I - \gamma A^T A)\vec{x}_i$

$\Rightarrow \vec{x}_n = (I - \gamma A^T A)^n \vec{x}_0$

(b) If we want $\vec{x}_n$ stable, then we need all eigenvalues of $-1 \le I - \gamma A^T A \le 1$.

Write $SVD(A) = U\Sigma V^T \qquad \Sigma = \text{diag}\,\sigma_i$

$I - \gamma A^T A = V I V^T - \gamma V\Sigma^T U^T U\Sigma V^T$

$= V(I - \gamma\Sigma^2)V^T$

So the eigenvalues of $I - \gamma A^T A$ is $1 - \gamma\sigma^2$

$-1 \le 1 - \gamma\sigma^2 \le 1 \Rightarrow 0 \le \gamma\sigma^2 \le 2 \Rightarrow \gamma \le \frac{2}{\max(\sigma)^2}$

(c) $f(\vec{x}) = \frac{1}{2}(A\vec{x} - \vec{b})^T(A\vec{x} - \vec{b}) = \frac{1}{2}\vec{x}^T A^T A\vec{x} - \vec{b}^T A\vec{x} + \vec{b}^T\vec{b}$

$\frac{\partial f(\vec{x})}{\partial \vec{x}} = \vec{x}^T A^T A - \vec{b}^T A \qquad \nabla f(\vec{x}) = A^T A\vec{x} - A^T\vec{b}$

$\varphi(\vec{x}) - \varphi(\vec{x}') = \vec{x} - \gamma\nabla f(\vec{x}) - (\vec{x}' - \gamma\nabla f(\vec{x}'))$

$= \vec{x} - \vec{x}' - \gamma A^T A(\vec{x} - \vec{x}')$

$= (I - \gamma A^T A)(\vec{x} - \vec{x}')$

$= V(I - \gamma\Sigma^2)V^T(\vec{x} - \vec{x}')$

$\|\varphi(\vec{x}) - \varphi(\vec{x}')\| = \|V(I - \gamma\Sigma^2)V^T(\vec{x} - \vec{x}')\|$

$= \|(I - \gamma\Sigma^2)(\vec{x} - \vec{x}')\|$

$= \left\| \begin{bmatrix} 1-\gamma\sigma_1^2 & & \\ & \ddots & \\ & & 1-\gamma\sigma_n^2 \end{bmatrix} (\vec{x} - \vec{x}') \right\|$ Here σ^2 = eigenvalue of $A^T A$

Let $\beta = \max\{|1 - \gamma\lambda_{\max}(A^T A)|, |1 - \gamma\lambda_{\min}(A^T A)|\}$

Then $\|\varphi(\vec{x}) - \varphi(\vec{x}')\| \le \beta\|\vec{x} - \vec{x}'\|$

(d) $\varphi(\vec{x}_k) - \varphi(\vec{x}^*) = \vec{x}_k - \gamma(A^TA\vec{x}_k - A^T\vec{b}) - (\vec{x}^* - \gamma(A^TA\vec{x}^* - A^T\vec{b}))$

$= \vec{x}_k - \gamma A^TA(\vec{x}_k - \vec{x}^*) - \vec{x}^*$ ①

$\vec{x}_{k+1} = \vec{x}_k - \gamma \nabla f(\vec{x}_k) = \vec{x}_k - \gamma(A^TA\vec{x}_k - A^T\vec{b})$ ②

$\nabla f(\vec{x}^*) = A^TA\vec{x}^* - A^T\vec{b} = 0 \Rightarrow A^T\vec{b} = A^TA\vec{x}^*$ ③

put ③ into ②, we get: $\vec{x}_{k+1} = \vec{x}_k - \gamma(A^TA\vec{x}_k - A^TA\vec{x}^*)$

$= \vec{x}_k - \gamma A^TA(\vec{x}_k - \vec{x}^*)$ ④

put ④ into ①, we get: $\varphi(\vec{x}_k) - \varphi(\vec{x}^*) = \vec{x}_{k+1} - \vec{x}^*$

so we prove that: $\|\vec{x}_{k+1} - \vec{x}^*\| = \|\varphi(\vec{x}_k) - \varphi(\vec{x}^*)\|$

From (c) we know: $\|\varphi(\vec{x}_k) - \varphi(\vec{x}^*)\| \le \beta \|\vec{x}_k - \vec{x}^*\| =$

$= \beta \|\varphi(\vec{x}_{k-1}) - \varphi(\vec{x}^*)\|$

$\le \beta^2 \|\vec{x}_{k-1} - \vec{x}^*\|$

$\le \beta^{k+1} \|\vec{x}_0 - \vec{x}^*\|$

so $\|\vec{x}_{k+1} - \vec{x}^*\| \le \beta^{k+1} \|\vec{x}_0 - \vec{x}^*\|$

(e) $f(\vec{x}) - f(\vec{x}^*) = \frac{1}{2}\vec{x}^TA^TA\vec{x} - \vec{b}^TA\vec{x} - \left(\frac{1}{2}\vec{x}^{*T}A^TA\vec{x}^* - \vec{b}^TA\vec{x}^*\right)$

$\frac{1}{2}\|A(\vec{x} - \vec{x}^*)\|^2 = \frac{1}{2}(\vec{x} - \vec{x}^*)^TA^TA(\vec{x} - \vec{x}^*)$

$= \frac{1}{2}\vec{x}^TA^TA\vec{x} - \vec{x}^TA^TA\vec{x}^* + \frac{1}{2}\vec{x}^{*T}A^TA\vec{x}^*$

$\Rightarrow f(\vec{x}) - f(\vec{x}^*) - \frac{1}{2}\|A(\vec{x} - \vec{x}^*)\|^2 = -\vec{b}^TA(\vec{x} - \vec{x}^*) - \vec{x}^{*T}A^TA\vec{x}^* - \vec{x}^TA^TA\vec{x}^*$

implement ③ $= -\vec{x}^{*T}A^TA(\vec{x} - \vec{x}^*) - \vec{x}^{*T}A^TA\vec{x}^* - \vec{x}^{*T}A^TA\vec{x}$

$= 0$

so we prove $f(\vec{x}) - f(\vec{x}^*) = \frac{1}{2}\|A(\vec{x} - \vec{x}^*)\|^2$

(f) $f(\vec{x}_k) - f(\vec{x}^*) = \frac{1}{2}\|A(\vec{x}_k - \vec{x}^*)\|^2$ let $\vec{y} = \vec{x}_k - \vec{x}^*$

$= \frac{1}{2} y^T A^T A y$

$= \frac{1}{2} y^T V \Sigma^2 V^T y$

$= \frac{1}{2} (V^T y)^T \begin{bmatrix} \sigma_1^2 & & \\ & \ddots & \\ & & \sigma_n^2 \end{bmatrix} (V^T y)$

$\leq \frac{\alpha}{2} \|V^T y\|^2$ where $\alpha = \lambda_{max}(A^T A)$

$= \frac{\alpha}{2} \|y\|^2$

$= \frac{\alpha}{2} \|\vec{x}_k - \vec{x}^*\|^2$

from (d), we know: $\|\vec{x}_k - \vec{x}^*\| \leq \beta^k \|\vec{x}_0 - \vec{x}^*\|$

So $\|\vec{x}_k - \vec{x}^*\|^2 \leq \beta^{2k} \|\vec{x}_0 - \vec{x}^*\|^2$

so $f(\vec{x}_k) - f(\vec{x}^*) \leq \frac{\alpha}{2} \beta^{2k} \|\vec{x}_0 - \vec{x}^*\|^2$

(g) $\beta = \max\{|1 - \gamma\lambda_{max}(A^T A)|, |1 - \gamma\lambda_{min}(A^T A)|\}$.

To make β as small as possible, we want $|1 - \gamma\lambda_{max}(A^T A)| = |1 - \gamma\lambda_{min}(A^T A)|$.

$\gamma\lambda_{max} - 1 = 1 - \gamma\lambda_{min}$

$\gamma = \frac{2}{\lambda_{max} + \lambda_{min}}$

$\beta = \left|1 - \frac{2\lambda_{max}}{\lambda_{max} + \lambda_{min}}\right| = \left|\frac{\lambda_{min} - \lambda_{max}}{\lambda_{max} + \lambda_{min}}\right| = \left|\frac{1 - \frac{\lambda_{max}}{\lambda_{min}}}{1 + \frac{\lambda_{max}}{\lambda_{min}}}\right| = \left|\frac{1-k}{1+k}\right|$

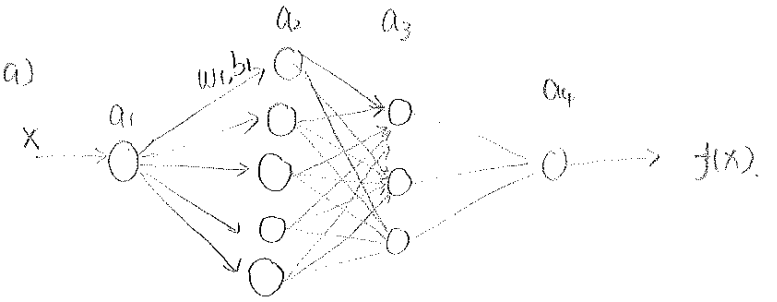
So $f(\vec{x}_k) - f(\vec{x}^*) = \frac{\alpha}{2} \left(\frac{k-1}{k+1}\right)^{2k} \|\vec{x}_0 - \vec{x}^*\|^2$

4. (a) $\frac{\partial L}{\partial a_1} = \sum_{i=1}^{7} 2\left(\sqrt{(a_i - x_1)^2 + (b_i - y_1)^2} - d_i\right) \cdot \frac{2(a_i - x_1) \cdot \frac{1}{2}}{2\sqrt{(a_i - x_1)^2 + (b_i - y_1)^2}}$

$= 2 \sum_{i=1}^{7} \left(1 - \frac{d_i}{\sqrt{(a_i - x_1)^2 + (b_i - y_1)^2}}\right)(a_i - x_1)$

$\frac{\partial L}{\partial y_1} = 2 \sum_{i=1}^{7} \left(1 - \frac{d_i}{\sqrt{(a_i - x_1)^2 + (b_i - y_1)^2}}\right)(b_i - y_1)$

5. (a)



layer 1	layer 2	layer 3	
$m_1 = n_1 = 1$	$m_2 = n_2 = 5$	$m_3 = n_3 = 3$	$a_4 : 1$
$a_1 : 1 \times 1$	$a_2 : 5 \times 1$	$a_3 : 3 \times 1$	
$w_1 : 5 \times 1$	$w_2 : 3 \times 5$	$w_3 : 1 \times 3$	
$b_1 : 5 \times 1$	$b_2 : 3 \times 1$	$b_3 : 1 \times 1$	

(b) $MSE(\hat{y}) = \frac{1}{2}(y - \hat{y})^T (y - \hat{y}) = \frac{1}{2}(y^T y - 2y^T \hat{y} + \hat{y}^T \hat{y})$

$\frac{\partial MSE(\hat{y})}{\partial(\hat{y})} = -y^T + \hat{y}^T$

(d) $\left(\frac{\partial MSE}{\partial a_i}\right)_l = \frac{\partial MSE}{(\partial a_i)_l} = \sum_k \frac{\partial MSE}{\partial (z_i)_k} \cdot \frac{\partial (z_i)_k}{\partial (a_i)_l}$

$= \sum_{kj} \frac{\partial MSE}{\partial (a_{i+1})_j} \cdot \frac{\partial (a_{i+1})_j}{\partial (z_i)_k} \cdot \frac{\partial (z_i)_k}{\partial (a_i)_l}$

$= \sum_{kj} \left(\frac{\partial MSE}{\partial a_{i+1}}\right)_j \cdot \frac{\partial \sigma(z_i)_j}{\partial (z_i)_k} \cdot \frac{\partial (w_i a_i + b_i)_k}{\partial (a_i)_l}$

$= \sum_{kj} \left(\frac{\partial MSE}{\partial a_{i+1}}\right)_j \cdot \delta_{jk} \left(\sigma'(z_i)\right)_k \cdot (w_i)_{kl}$

4. (b) see code and results below. 0.01, 0.001, 0.0001 are reasonable steps.

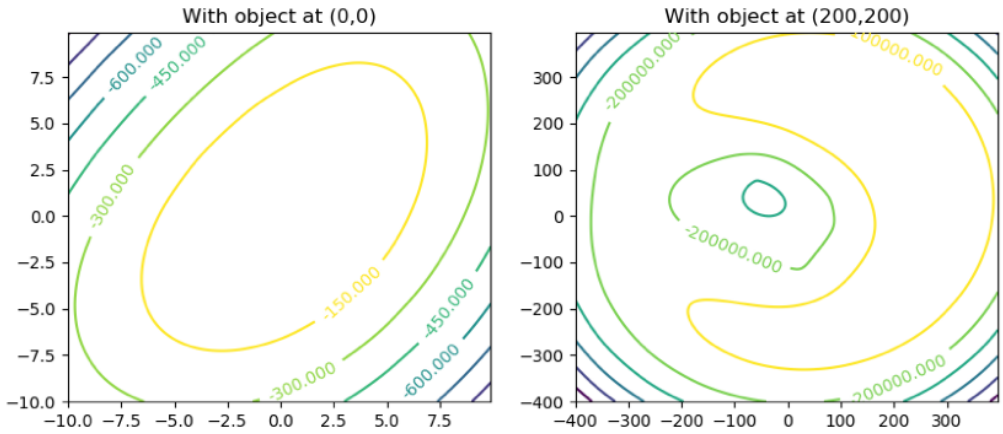
```
lr= 1
The real object location is
[[44.38632327 33.36743274]]
/Users/siyao/Desktop/6_ml/hw7/hw07-data/sensor_location1_starter/part_b_starter.py:34: RuntimeWarning: overflow enc
ountered in square
  temp2 = np.sqrt((sensor_loc.T[0] - x)**2 + (sensor_loc.T[1] - y)**2)
/Users/siyao/Desktop/6_ml/hw7/hw07-data/sensor_location1_starter/part_b_starter.py:35: RuntimeWarning: overflow enc
ountered in double_scalars
  grad[0] = 2 * np.sum((1 - single_distance / temp2)* temp1)
/anaconda3/lib/python3.6/site-packages/numpy/core/_methods.py:32: RuntimeWarning: overflow encountered in reduce
  return umr_sum(a, axis, dtype, out, keepdims)
/Users/siyao/Desktop/6_ml/hw7/hw07-data/sensor_location1_starter/part_b_starter.py:62: RuntimeWarning: invalid valu
e encountered in subtract
  obj_loc -= grad*lr
The estimated object location with zero initialization is
[nan nan]
The estimated object location with random initialization is
[nan nan]
```

```
In [70]: run part_b_starter.py
lr= 0.01
The real object location is
[[44.38632327 33.36743274]]
The estimated object location with zero initialization is
[43.07188433 32.71217817]
The estimated object location with random initialization is
[43.07188433 32.71217817]
```

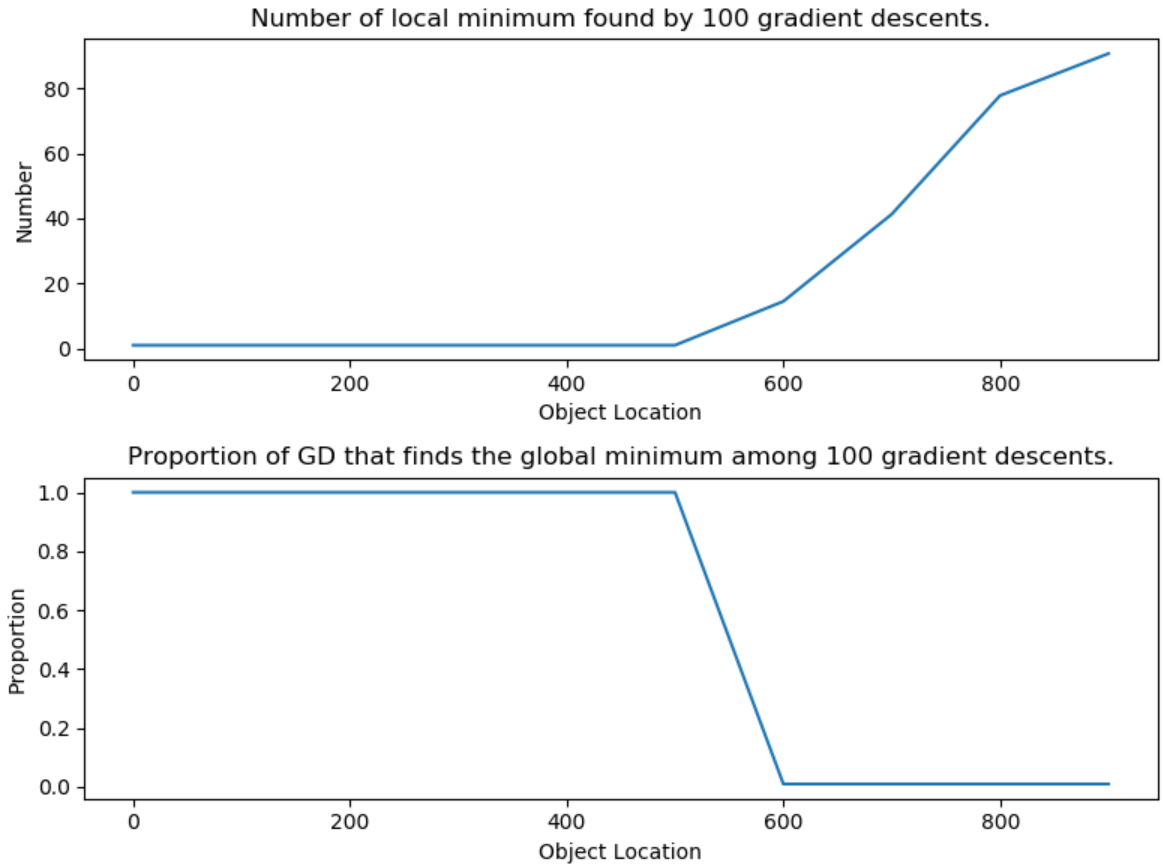
```
In [71]: run part_b_starter.py
lr= 0.001
The real object location is
[[44.38632327 33.36743274]]
The estimated object location with zero initialization is
[43.07188433 32.71217817]
The estimated object location with random initialization is
[43.07188433 32.71217817]
```

```
In [74]: run part_b_starter.py
lr= 0.0001
The real object location is
[[44.38632327 33.36743274]]
The estimated object location with zero initialization is
[42.56743732 32.03587615]
The estimated object location with random initialization is
[42.57881496 32.05165343]
```

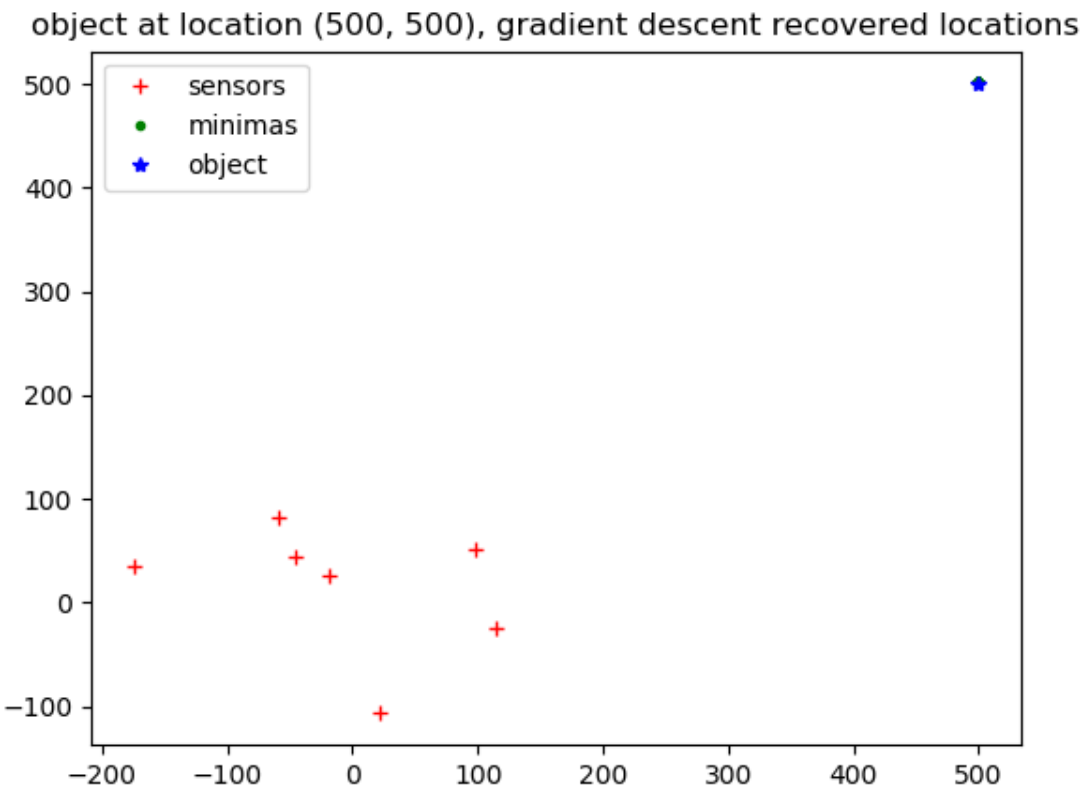
4 (c) counter plot of log likelihood



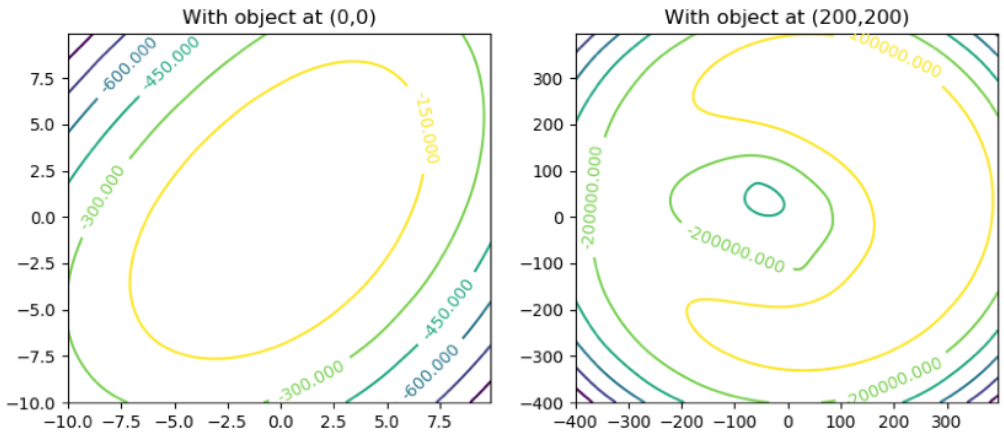
number of local minimum/average portion against x1



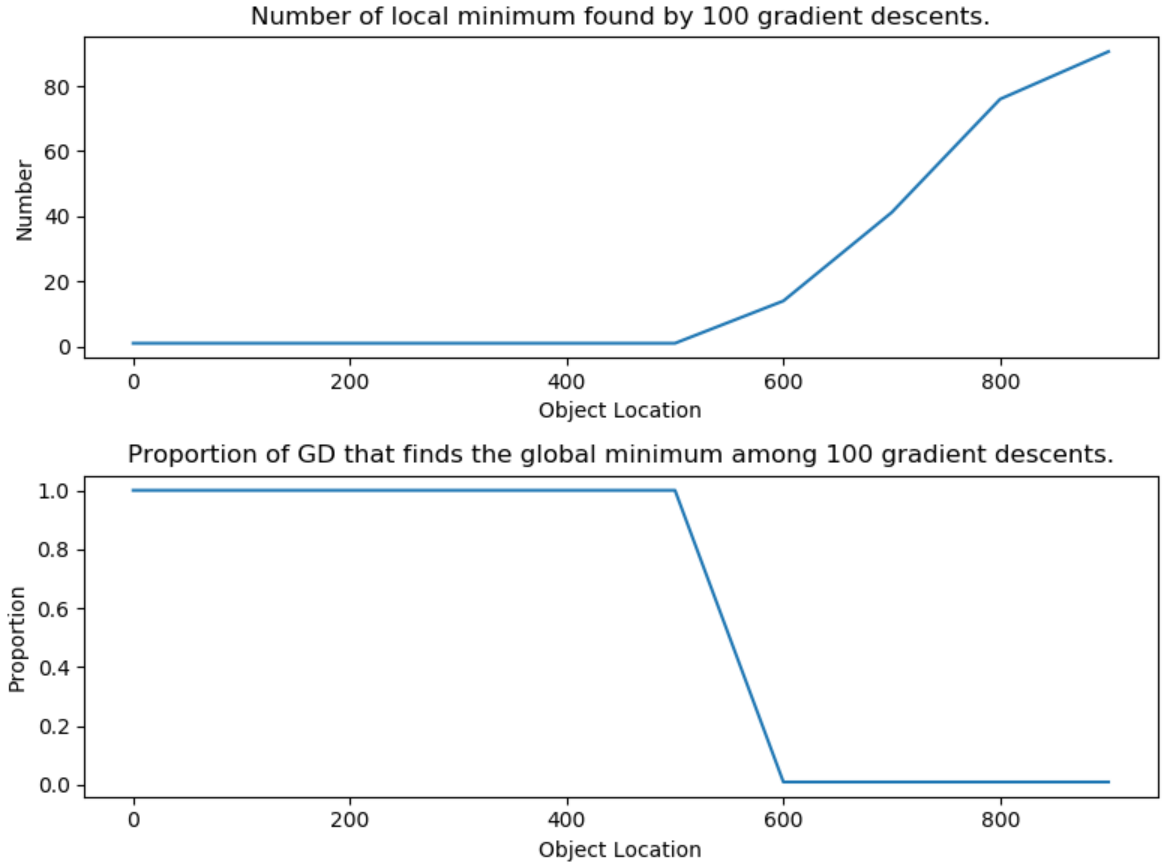
sensor loc, true obj loc, MLE obj loc.
We can see that all 100 MEL obj loc are almost all at true obj loc.



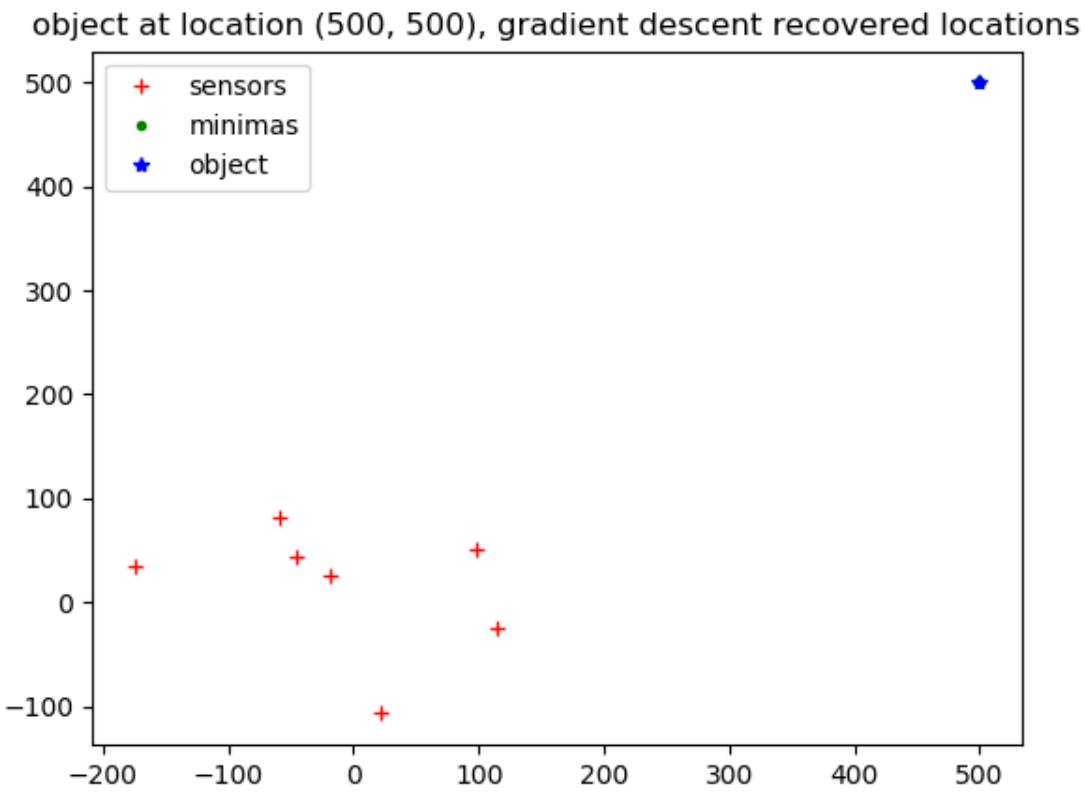
4 (d) counter plot of log likelihood



number of local minimum/average portion against x1

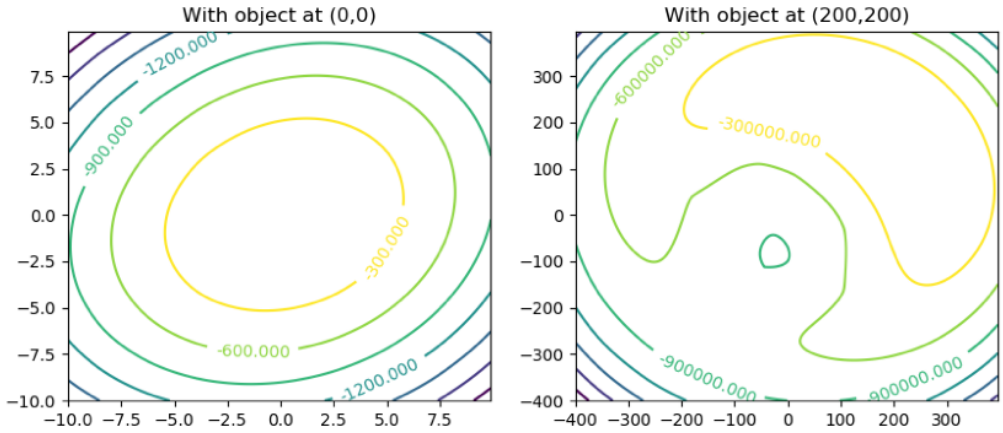


sensor loc, true obj loc, MLE obj loc.
We can see that all 100 MEL obj loc are almost all at true obj loc.

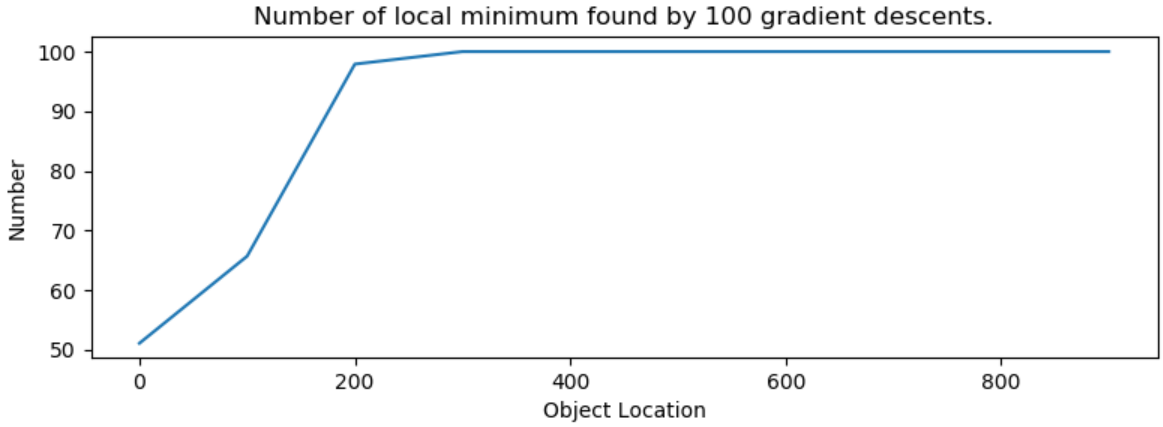


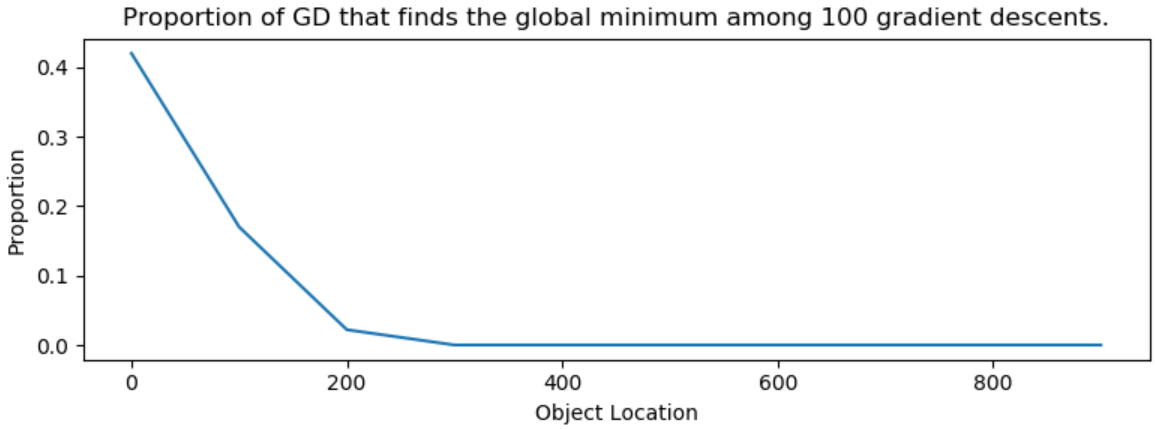
So we can see that after reducing the measurement noise, we get similar answer as with larger noise.

4 (e) counter plot of log likelihood

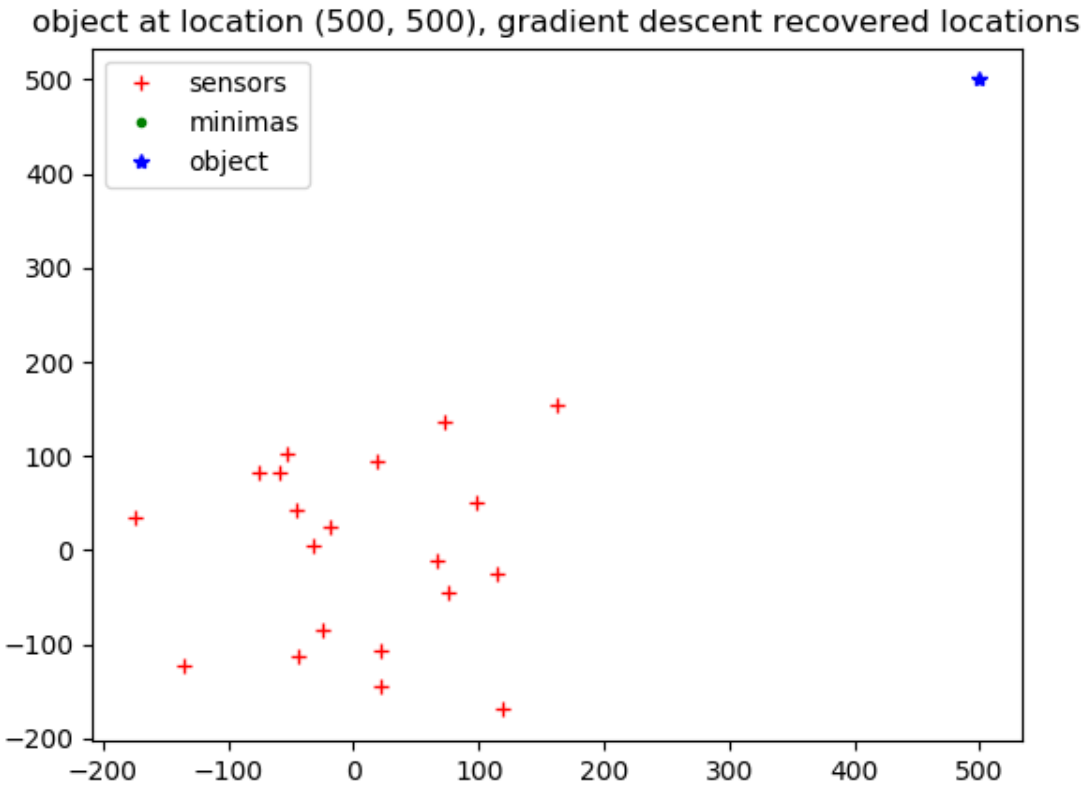


number of local minimum/average portion against x1





sensor loc, true obj loc, MLE obj loc.
We can see that all 100 MEL obj loc are almost all at true obj loc.



When increasing number of sensor from 7 to 20, the number of local minimum increases much earlier, meaning even when our object location is still relatively close to initial position, we already get many local minimum.
However, the counter of likelihood is restricted in a smaller region, so we have a better estimate of object location. So increasing tensor number helps us get a better estimate of true location.

4 (f)

```
In [103]: run part_f_starter.py
The real sensor locations are
[[176.4052346   40.01572084]
 [ 97.87379841 224.08931992]
 [186.75579901 -97.72778799]
 [ 95.00884175 -15.13572083]
 [-10.32188518  41.05985019]
 [ 14.40435712 145.4273507 ]
 [ 76.10377251  12.16750165]]
The predicted sensor locations are
[[176.28223286  39.76550241]
 [ 97.73185793 224.12127842]
 [186.68461118 -97.59207958]
 [ 94.90639305 -15.16741152]
 [-10.17511568  40.93691838]
 [ 14.19497946 145.42410633]
 [ 76.11863016  12.24807699]]
The MSE for Case 1 is 301.60439216914386
The MSE for Case 2 is 6127.083605044189
The MSE for Case 2 (if we knew mu is [300,300]) is 448.04338939127314
```

common

```
import numpy as np
import scipy.spatial
import matplotlib
import matplotlib.pyplot as plt
#####################################################################
########  Data Generating Functions ##################################
#####################################################################
def generate_sensors(k = 7, d = 2):
    """
    Generate sensor locations.
    Input:
    k: The number of sensors.
    d: The spatial dimension.
    Output:
    sensor_loc: k * d numpy array.
    """
    sensor_loc = 100*np.random.randn(k,d)
    return sensor_loc

def generate_data(sensor_loc, k = 7, d = 2,
                                    n = 1, original_dist = True, sigma_s = 100):
    """
    Generate the locations of n points and distance measurements.

    Input:
    sensor_loc: k * d numpy array. Location of sensor.
    k: The number of sensors.
    d: The spatial dimension.
    n: The number of points.
    original_dist: Whether the data are generated from the original
    distribution.
    sigma_s: the standard deviation of the distribution
    that generate each object location.

    Output:
    obj_loc: n * d numpy array. The location of the n objects.
    distance: n * k numpy array. The distance between object and
    the k sensors.
    """
    assert k, d == sensor_loc.shape

    obj_loc = sigma_s*np.random.randn(n, d)
    if not original_dist:
            obj_loc = sigma_s*np.random.randn(n, d)+([300,300])

    distance = scipy.spatial.distance.cdist(obj_loc, sensor_loc, metric='euclidean')
    distance += np.random.randn(n, k)
    return obj_loc, distance

def generate_data_given_location(sensor_loc, obj_loc, k = 7, d = 2, dis_noise=1):
    """
    Generate the distance measurements given location of a single object and sensor.

    Input:
    obj_loc: 1 * d numpy array. Location of object
    sensor_loc: k * d numpy array. Location of sensor.
    k: The number of sensors.
    d: The spatial dimension.

    Output:
    distance: 1 * k numpy array. The distance between object and
    the k sensors.
    """
    assert k, d == sensor_loc.shape

    distance = scipy.spatial.distance.cdist(obj_loc, sensor_loc, metric='euclidean')
    distance += np.random.randn(1, k) * dis_noise
    return obj_loc, distance
```

4b


```
from common import *
import pdb

###########################################################################
######## Part b ###################################
###########################################################################

###########################################################################
#########  Gradient Computing and MLE ###################################
###########################################################################
def compute_gradient_of_likelihood(single_obj_loc, sensor_loc, single_distance):
    """
    Compute the gradient of the loglikelihood function for part a.

    Input:
    single_obj_loc: 1 * d numpy array.
    Location of the single object.

    sensor_loc: k * d numpy array.
    Location of sensor.

    single_distance: k dimensional numpy array.
    Observed distance of the object.

    Output:
    grad: d-dimensional numpy array.

    """
    grad = np.zeros_like(single_obj_loc)
    #Your code: implement the gradient of loglikelihood
    x = single_obj_loc[0]
    y = single_obj_loc[1]
    temp1 = sensor_loc.T[0]-x
    temp2 = np.sqrt((sensor_loc.T[0] - x)**2 + (sensor_loc.T[1] - y)**2)
    grad[0] = 2 * np.sum((1 - single_distance / temp2)* temp1)
    temp3 = sensor_loc.T[1]-y
    grad[1] = 2 * np.sum((1 - single_distance / temp2)* temp3)
    return grad*(-1.0)

def find_mle_by_grad_descent_part_b(initial_obj_loc, sensor_loc, single_distance,
        lr=0.001, num_iters = 10000):
    """
    Compute the gradient of the loglikelihood function for part a.

    Input:
    initial_obj_loc: 1 * d numpy array.
    Initialized Location of the single object.

    sensor_loc: k * d numpy array. Location of sensor.

    single_distance: k dimensional numpy array.
    Observed distance of the object.

    Output:
    obj_loc: 1 * d numpy array. The mle for the location of the object.

    """
    obj_loc = initial_obj_loc[0]
    # Your code: do gradient descent
    for i in range(num_iters):
        grad = compute_gradient_of_likelihood(obj_loc, sensor_loc, single_distance)
        obj_loc -= grad*lr

    return obj_loc

if __name__ == "__main__":
    ###########################################################################
    #########  MAIN ###########################################################
    ###########################################################################

    # Your code: set some appropriate learning rate here
    lr = 1
    print('lr=',lr)
```

```
np.random.seed(0)
sensor_loc = generate_sensors()
obj_loc, distance = generate_data(sensor_loc)
single_distance = distance[0]
print('The real object location is')
print(obj_loc)
# Initialized as [0,0]
initial_obj_loc = np.array([[0.,0.]])
estimated_obj_loc = find_mle_by_grad_descent_part_b(initial_obj_loc,
                    sensor_loc, single_distance, lr=lr, num_iters = 10000)
print('The estimated object location with zero initialization is')
print(estimated_obj_loc)

# Random initialization.
initial_obj_loc = np.random.randn(1,2)
estimated_obj_loc = find_mle_by_grad_descent_part_b(initial_obj_loc,
                    sensor_loc, single_distance, lr=lr, num_iters = 10000)
print('The estimated object location with random initialization is')
print(estimated_obj_loc)
```

```
from common import *
from part_b_starter import find_mle_by_grad_descent_part_b
import pdb

#########################################################################
######### Part c ########################################################
#########################################################################
def log_likelihood(obj_loc, sensor_loc, distance):
  """
  This function computes the log likelihood (as expressed in Part a).
  Input:
  obj_loc: shape [1,2]
  sensor_loc: shape [7,2]
  distance: shape [7]
  Output:
  The log likelihood function value.
  """
  # Your code: compute the log likelihood
  x = obj_loc[0][0]
  y = obj_loc[0][1]
  func_value = 0.0
  func_value = - np.sum((np.sqrt((sensor_loc.T[0] - x)**2 + (sensor_loc.T[1] - y)**2
) - distance)**2)

  return func_value

if __name__ == "__main__":
  ########################################################################
  ######### Compute the function value at local minimum for all experiments.###
  ########################################################################
  num_sensors = 7

  np.random.seed(100)
  sensor_loc = generate_sensors(k=num_sensors)

  # num_data_replicates = 10
  num_gd_replicates = 100

  obj_locs = [[[i,i]] for i in np.arange(0,1000,100)]

  func_values = np.zeros((len(obj_locs),10, num_gd_replicates))
  # record sensor_loc, obj_loc, 100 found minimas
  minimas = np.zeros((len(obj_locs), 10, num_gd_replicates, 2))
  true_object_locs = np.zeros((len(obj_locs), 10, 2))

  for i, obj_loc in enumerate(obj_locs):
      for j in range(10):
          obj_loc, distance = generate_data_given_location(sensor_loc, obj_loc,
                                                           k = num_sensors, d = 2)
          true_object_locs[i, j, :] = np.array(obj_loc)

          for gd_replicate in range(num_gd_replicates):
              initial_obj_loc = np.random.randn(1,2)* (100 * i+1)
              obj_loc = find_mle_by_grad_descent_part_b(initial_obj_loc,
                        sensor_loc, distance[0], lr=0.1, num_iters = 1000)
              minimas[i, j, gd_replicate, :] = np.array(obj_loc)
              func_value = log_likelihood(obj_loc, sensor_loc, distance[0])
              func_values[i, j, gd_replicate] = func_value

  np.save('func_values.npy', func_values)

  func_values = np.load('func_values.npy')
  ########################################################################
  ######### Calculate the things to be plotted. ###
  ########################################################################
  local_mins = [[np.unique(func_values[i,j].round(decimals=2)) for j in range(10)] f
or i in range(10)]
  num_local_min = [[len(local_mins[i][j]) for j in range(10)] for i in range(10)]
  proportion_global = [[sum(func_values[i,j].round(decimals=2) == min(local_mins[i][
j]))*1.0/100 \
                        for j in range(10)] for i in range(10)]
```

```
  num_local_min = np.array(num_local_min)
  num_local_min = np.mean(num_local_min, axis = 1)

  proportion_global = np.array(proportion_global)
  proportion_global = np.mean(proportion_global, axis = 1)

  ######################################################################
  ######### Plots. #####################################################
  ######################################################################
  fig, axes = plt.subplots(figsize=(8,6), nrows=2, ncols=1)
  fig.tight_layout()
  plt.subplot(211)

  plt.plot(np.arange(0,1000,100), num_local_min)
  plt.title('Number of local minimum found by 100 gradient descents.')
  plt.xlabel('Object Location')
  plt.ylabel('Number')
  #plt.savefig('num_obj.png')
  # Proportion of gradient descents that find the local minimum of minimum value.

  plt.subplot(212)
  plt.plot(np.arange(0,1000,100), proportion_global)
  plt.title('Proportion of GD that finds the global minimum among 100 gradient desce
nts.')
  plt.xlabel('Object Location')
  plt.ylabel('Proportion')
  fig.tight_layout()
  plt.savefig('prop_obj.png')

  ######################################################################
  ######### Plots of contours. #########################################
  ######################################################################
  np.random.seed(0)
  # sensor_loc = np.random.randn(7,2) * 10
  x = np.arange(-10.0, 10.0, 0.1)
  y = np.arange(-10.0, 10.0, 0.1)
  X, Y = np.meshgrid(x, y)
  obj_loc = [[0,0]]
  obj_loc, distance = generate_data_given_location(sensor_loc, obj_loc, k = num_sens
ors, d = 2)

  Z =  np.array([[log_likelihood([[X[i,j],Y[i,j]]],
                  sensor_loc, distance[0]) for j in range(len(X))] \
                for i in range(len(X))])


  plt.figure(figsize=(10,4))
  plt.subplot(121)
  CS = plt.contour(X, Y, Z)
  plt.clabel(CS, inline=1, fontsize=10)
  plt.title('With object at (0,0)')
  #plt.show()

  np.random.seed(0)
  # sensor_loc = np.random.randn(7,2) * 10
  x = np.arange(-400,400, 4)
  y = np.arange(-400,400, 4)
  X, Y = np.meshgrid(x, y)
  obj_loc = [[200,200]]
  obj_loc, distance = generate_data_given_location(sensor_loc,
                        obj_loc, k = num_sensors, d = 2)

  Z =  np.array([[log_likelihood([[X[i,j],Y[i,j]]],
                    sensor_loc, distance[0]) for j in range(len(X))] \
                for i in range(len(X))])


  # Create a simple contour plot with labels using default colors.  The
  # inline argument to clabel will control whether the labels are draw
  # over the line segments of the contour, removing the lines beneath
  # the label
  #plt.figure()
  plt.subplot(122)
```

```
from common import *
from part_b_starter import find_mle_by_grad_descent_part_b
import pdb

###########################################################################
######### Part c ##########################################################
###########################################################################
def log_likelihood(obj_loc, sensor_loc, distance):
  """
  This function computes the log likelihood (as expressed in Part a).
  Input:
  obj_loc: shape [1,2]
  sensor_loc: shape [7,2]
  distance: shape [7]
  Output:
  The log likelihood function value.
  """
  # Your code: compute the log likelihood
  x = obj_loc[0][0]
  y = obj_loc[0][1]
  func_value = 0.0
  func_value = - np.sum((np.sqrt((sensor_loc.T[0] - x)**2 + (sensor_loc.T[1] - y)**2
) - distance)**2)

  return func_value

if __name__ == "__main__":
  ###########################################################################
  ######### Compute the function value at local minimum for all experiments.###
  ###########################################################################
  num_sensors = 7

  np.random.seed(100)
  sensor_loc = generate_sensors(k=num_sensors)

  # num_data_replicates = 10
  num_gd_replicates = 100

  obj_locs = [[[i,i]] for i in np.arange(0,1000,100)]

  func_values = np.zeros((len(obj_locs),10, num_gd_replicates))
  # record sensor_loc, obj_loc, 100 found minimas
  minimas = np.zeros((len(obj_locs), 10, num_gd_replicates, 2))
  true_object_locs = np.zeros((len(obj_locs), 10, 2))

  for i, obj_loc in enumerate(obj_locs):
      for j in range(10):
          obj_loc, distance = generate_data_given_location(sensor_loc, obj_loc,
                                                           k = num_sensors, d = 2, d
is_noise=0.01)
          true_object_locs[i, j, :] = np.array(obj_loc)

          for gd_replicate in range(num_gd_replicates):
              initial_obj_loc = np.random.randn(1,2)* (100 * i+1)
              obj_loc = find_mle_by_grad_descent_part_b(initial_obj_loc,
                          sensor_loc, distance[0], lr=0.1, num_iters = 1000)
              minimas[i, j, gd_replicate, :] = np.array(obj_loc)
              func_value = log_likelihood(obj_loc, sensor_loc, distance[0])
              func_values[i, j, gd_replicate] = func_value

  np.save('func_values.npy', func_values)

  func_values = np.load('func_values.npy')
  ###########################################################################
  ######### Calculate the things to be plotted. ###
  ###########################################################################
  local_mins = [[np.unique(func_values[i,j].round(decimals=2)) for j in range(10)] f
or i in range(10)]
  num_local_min = [[len(local_mins[i][j]) for j in range(10)] for i in range(10)]
  proportion_global = [[sum(func_values[i,j].round(decimals=2) == min(local_mins[i][
j]))*1.0/100 \
                        for j in range(10)] for i in range(10)]
```

```
  num_local_min = np.array(num_local_min)
  num_local_min = np.mean(num_local_min, axis = 1)

  proportion_global = np.array(proportion_global)
  proportion_global = np.mean(proportion_global, axis = 1)

  ######################################################################
  ######### Plots. #####################################################
  ######################################################################
  fig, axes = plt.subplots(figsize=(8,6), nrows=2, ncols=1)
  fig.tight_layout()
  plt.subplot(211)

  plt.plot(np.arange(0,1000,100), num_local_min)
  plt.title('Number of local minimum found by 100 gradient descents.')
  plt.xlabel('Object Location')
  plt.ylabel('Number')
  #plt.savefig('num_obj.png')
  # Proportion of gradient descents that find the local minimum of minimum value.

  plt.subplot(212)
  plt.plot(np.arange(0,1000,100), proportion_global)
  plt.title('Proportion of GD that finds the global minimum among 100 gradient desce
nts.')
  plt.xlabel('Object Location')
  plt.ylabel('Proportion')
  fig.tight_layout()
  plt.savefig('prop_obj.png')

  ######################################################################
  ######### Plots of contours. #########################################
  ######################################################################
  np.random.seed(0)
  # sensor_loc = np.random.randn(7,2) * 10
  x = np.arange(-10.0, 10.0, 0.1)
  y = np.arange(-10.0, 10.0, 0.1)
  X, Y = np.meshgrid(x, y)
  obj_loc = [[0,0]]
  obj_loc, distance = generate_data_given_location(sensor_loc, obj_loc, k = num_sens
ors, d = 2, dis_noise=0.01)

  Z =  np.array([[log_likelihood([[X[i,j],Y[i,j]]],
                  sensor_loc, distance[0]) for j in range(len(X))] \
                for i in range(len(X))])


  plt.figure(figsize=(10,4))
  plt.subplot(121)
  CS = plt.contour(X, Y, Z)
  plt.clabel(CS, inline=1, fontsize=10)
  plt.title('With object at (0,0)')
  #plt.show()

  np.random.seed(0)
  # sensor_loc = np.random.randn(7,2) * 10
  x = np.arange(-400,400, 4)
  y = np.arange(-400,400, 4)
  X, Y = np.meshgrid(x, y)
  obj_loc = [[200,200]]
  obj_loc, distance = generate_data_given_location(sensor_loc,
                        obj_loc, k = num_sensors, d = 2, dis_noise=0.01)

  Z =  np.array([[log_likelihood([[X[i,j],Y[i,j]]],
                    sensor_loc, distance[0]) for j in range(len(X))] \
                for i in range(len(X))])


  # Create a simple contour plot with labels using default colors.  The
  # inline argument to clabel will control whether the labels are draw
  # over the line segments of the contour, removing the lines beneath
  # the label
  #plt.figure()
```

```
  plt.subplot(122)
  CS = plt.contour(X, Y, Z)
  plt.clabel(CS, inline=1, fontsize=10)
  plt.title('With object at (200,200)')
  #plt.show()
  plt.savefig('likelihood_landscape.png')


  ########################################################################
  ######### Plots of Found local minimas. ################################################
#
  ########################################################################
  #sensor_loc
  #minimas = np.zeros((len(obj_locs), 10, num_gd_replicates, 2))
  #true_object_locs = np.zeros((len(obj_locs), 10, 2))
  object_loc_i = 5
  trail = 0

  plt.figure()
  plt.plot(sensor_loc[:, 0], sensor_loc[:, 1], 'r+', label="sensors")
  plt.plot(minimas[object_loc_i, trail, :, 0], minimas[object_loc_i, trail, :, 1], '
g.', label="minimas")
  plt.plot(true_object_locs[object_loc_i, trail, 0], true_object_locs[object_loc_i,
trail, 1], 'b*', label="object")
  plt.title('object at location (%d, %d), gradient descent recovered locations' % (o
bject_loc_i*100, object_loc_i*100))
  plt.legend()
  plt.savefig('2D_vis.png')
```

```
from common import *
from part_b_starter import find_mle_by_grad_descent_part_b
import pdb

###########################################################################
######### Part c ##########################################################
###########################################################################
def log_likelihood(obj_loc, sensor_loc, distance):
  """
  This function computes the log likelihood (as expressed in Part a).
  Input:
  obj_loc: shape [1,2]
  sensor_loc: shape [7,2]
  distance: shape [7]
  Output:
  The log likelihood function value.
  """
  # Your code: compute the log likelihood
  x = obj_loc[0][0]
  y = obj_loc[0][1]
  func_value = 0.0
  func_value = - np.sum((np.sqrt((sensor_loc.T[0] - x)**2 + (sensor_loc.T[1] - y)**2
) - distance)**2)

  return func_value

if __name__ == "__main__":
  ###########################################################################
  ######### Compute the function value at local minimum for all experiments.###
  ###########################################################################
  num_sensors = 20

  np.random.seed(100)
  sensor_loc = generate_sensors(k=num_sensors)

  # num_data_replicates = 10
  num_gd_replicates = 100

  obj_locs = [[[i,i]] for i in np.arange(0,1000,100)]

  func_values = np.zeros((len(obj_locs),10, num_gd_replicates))
  # record sensor_loc, obj_loc, 100 found minimas
  minimas = np.zeros((len(obj_locs), 10, num_gd_replicates, 2))
  true_object_locs = np.zeros((len(obj_locs), 10, 2))

  for i, obj_loc in enumerate(obj_locs):
      for j in range(10):
          obj_loc, distance = generate_data_given_location(sensor_loc, obj_loc,
                                                           k = num_sensors, d = 2, d
is_noise=1)
          true_object_locs[i, j, :] = np.array(obj_loc)

          for gd_replicate in range(num_gd_replicates):
              initial_obj_loc = np.random.randn(1,2)* (100 * i+1)
              obj_loc = find_mle_by_grad_descent_part_b(initial_obj_loc,
                          sensor_loc, distance[0], lr=0.1, num_iters = 1000)
              minimas[i, j, gd_replicate, :] = np.array(obj_loc)
              func_value = log_likelihood(obj_loc, sensor_loc, distance[0])
              func_values[i, j, gd_replicate] = func_value

  np.save('func_values.npy', func_values)

  func_values = np.load('func_values.npy')
  ###########################################################################
  ######### Calculate the things to be plotted. ###
  ###########################################################################
  local_mins = [[np.unique(func_values[i,j].round(decimals=2)) for j in range(10)] f
or i in range(10)]
  num_local_min = [[len(local_mins[i][j]) for j in range(10)] for i in range(10)]
  proportion_global = [[sum(func_values[i,j].round(decimals=2) == min(local_mins[i][
j]))*1.0/100 \
                        for j in range(10)] for i in range(10)]
```

```
  num_local_min = np.array(num_local_min)
  num_local_min = np.mean(num_local_min, axis = 1)

  proportion_global = np.array(proportion_global)
  proportion_global = np.mean(proportion_global, axis = 1)

  ######################################################################
  ######### Plots. #####################################################
  ######################################################################
  fig, axes = plt.subplots(figsize=(8,6), nrows=2, ncols=1)
  fig.tight_layout()
  plt.subplot(211)

  plt.plot(np.arange(0,1000,100), num_local_min)
  plt.title('Number of local minimum found by 100 gradient descents.')
  plt.xlabel('Object Location')
  plt.ylabel('Number')
  #plt.savefig('num_obj.png')
  # Proportion of gradient descents that find the local minimum of minimum value.

  plt.subplot(212)
  plt.plot(np.arange(0,1000,100), proportion_global)
  plt.title('Proportion of GD that finds the global minimum among 100 gradient desce
nts.')
  plt.xlabel('Object Location')
  plt.ylabel('Proportion')
  fig.tight_layout()
  plt.savefig('prop_obj.png')

  ######################################################################
  ######### Plots of contours. #########################################
  ######################################################################
  np.random.seed(0)
  # sensor_loc = np.random.randn(7,2) * 10
  x = np.arange(-10.0, 10.0, 0.1)
  y = np.arange(-10.0, 10.0, 0.1)
  X, Y = np.meshgrid(x, y)
  obj_loc = [[0,0]]
  obj_loc, distance = generate_data_given_location(sensor_loc, obj_loc, k = num_sens
ors, d = 2, dis_noise=1)

  Z =  np.array([[log_likelihood([[X[i,j],Y[i,j]]],
                  sensor_loc, distance[0]) for j in range(len(X))] \
                for i in range(len(X))])


  plt.figure(figsize=(10,4))
  plt.subplot(121)
  CS = plt.contour(X, Y, Z)
  plt.clabel(CS, inline=1, fontsize=10)
  plt.title('With object at (0,0)')
  #plt.show()

  np.random.seed(0)
  # sensor_loc = np.random.randn(7,2) * 10
  x = np.arange(-400,400, 4)
  y = np.arange(-400,400, 4)
  X, Y = np.meshgrid(x, y)
  obj_loc = [[200,200]]
  obj_loc, distance = generate_data_given_location(sensor_loc,
                        obj_loc, k = num_sensors, d = 2, dis_noise=1)

  Z =  np.array([[log_likelihood([[X[i,j],Y[i,j]]],
                    sensor_loc, distance[0]) for j in range(len(X))] \
                for i in range(len(X))])


  # Create a simple contour plot with labels using default colors.  The
  # inline argument to clabel will control whether the labels are draw
  # over the line segments of the contour, removing the lines beneath
  # the label
  #plt.figure()
```

4e


```
  plt.subplot(122)
  CS = plt.contour(X, Y, Z)
  plt.clabel(CS, inline=1, fontsize=10)
  plt.title('With object at (200,200)')
  #plt.show()
  plt.savefig('likelihood_landscape.png')


  ##########################################################################
  ######### Plots of Found local minimas. ##################################################
#
  ##########################################################################
  #sensor_loc
  #minimas = np.zeros((len(obj_locs), 10, num_gd_replicates, 2))
  #true_object_locs = np.zeros((len(obj_locs), 10, 2))
  object_loc_i = 5
  trail = 0

  plt.figure()
  plt.plot(sensor_loc[:, 0], sensor_loc[:, 1], 'r+', label="sensors")
  plt.plot(minimas[object_loc_i, trail, :, 0], minimas[object_loc_i, trail, :, 1], '
g.', label="minimas")
  plt.plot(true_object_locs[object_loc_i, trail, 0], true_object_locs[object_loc_i,
trail, 1], 'b*', label="object")
  plt.title('object at location (%d, %d), gradient descent recovered locations' % (o
bject_loc_i*100, object_loc_i*100))
  plt.legend()
  plt.savefig('2D_vis.png')
```

4f


```
from common import *
from part_b_starter import find_mle_by_grad_descent_part_b
from part_b_starter import compute_gradient_of_likelihood
from part_c_starter import log_likelihood
import pdb

########################################################################
#########  Gradient Computing and MLE ##################################
########################################################################
def compute_grad_likelihood(sensor_loc, obj_loc, distance):
    """
    Compute the gradient of the loglikelihood function for part f.

    Input:
    sensor_loc: k * d numpy array.
    Location of sensors.

    obj_loc: n * d numpy array.
    Location of the objects.

    distance: n * k dimensional numpy array.
    Observed distance of the object.

    Output:
    grad: k * d numpy array.
    """
    grad = np.zeros(sensor_loc.shape)
    # Your code: finish the grad loglike
    for i, sensor in enumerate(sensor_loc):
        grad[i] =  compute_gradient_of_likelihood([sensor], obj_loc, distance.T[i])
    return grad

def find_mle_by_grad_descent(initial_sensor_loc,
                obj_loc, distance, lr=0.001, num_iters = 1000):
    """
    Compute the gradient of the loglikelihood function for part f.

    Input:
    initial_sensor_loc: k * d numpy array.
    Initialized Location of the sensors.

    obj_loc: n * d numpy array. Location of the n objects.

    distance: n * k dimensional numpy array.
    Observed distance of the n object.

    Output:
    sensor_loc: k * d numpy array. The mle for the location of the object.

    """
    sensor_loc = initial_sensor_loc
    # Your code: finish the gradient descent
    for i in range(num_iters):
        grad = compute_grad_likelihood(sensor_loc, obj_loc, distance)
        sensor_loc -= grad*lr
    return sensor_loc

########################################################################
#########  Gradient Computing and MLE ##################################
########################################################################

np.random.seed(0)
sensor_loc = generate_sensors()
obj_loc, distance = generate_data(sensor_loc, n = 100)
print('The real sensor locations are')
print(sensor_loc)
# Initialized as zeros.
initial_sensor_loc = np.zeros((7,2)) #np.random.randn(7,2)
estimated_sensor_loc = find_mle_by_grad_descent(initial_sensor_loc,
                    obj_loc, distance, lr=0.001, num_iters = 1000)
print('The predicted sensor locations are')
print(estimated_sensor_loc)
```

```
##########################################################################
#########  Estimate distance given estimated sensor locations. #########
##########################################################################

def compute_distance_with_sensor_and_obj_loc(sensor_loc, obj_loc):
    """
    stimate distance given estimated sensor locations.

    Input:
    sensor_loc: k * d numpy array.
    Location of the sensors.

    obj_loc: n * d numpy array. Location of the n objects.

    Output:
    distance: n * k dimensional numpy array.
    """
    estimated_distance = scipy.spatial.distance.cdist(obj_loc, sensor_loc, metric='e
uclidean')
    return estimated_distance

##########################################################################
#########  MAIN  #########################################################
##########################################################################
np.random.seed(100)
##########################################################################
#########  Case 1. #######################################################
##########################################################################

mse = 100000
obj_loc, distance = generate_data(sensor_loc, k = 7, d = 2, n = 100, original_dist =
 True)
for i in range(100):
    # Your code: compute the mse for this case
    est_obj_loc = find_mle_by_grad_descent(np.random.randn(100,2), estimated_sensor_
loc, distance.T)
    diff = obj_loc - est_obj_loc
    mse_i = np.linalg.norm(diff)
    mse = min(mse, mse_i)

print('The MSE for Case 1 is {}'.format(mse))

##########################################################################
#########  Case 2. #######################################################
##########################################################################
mse = 10000

obj_loc, distance = generate_data(sensor_loc, k = 7, d = 2, n = 100, original_dist =
 False)
for i in range(100):
    # Your code: compute the mse for this case
    est_obj_loc = find_mle_by_grad_descent(np.random.randn(100,2), estimated_sensor_
loc, distance.T)
    diff = obj_loc - est_obj_loc
    mse_i = np.linalg.norm(diff)
    mse = min(mse, mse_i)

print('The MSE for Case 2 is {}'.format(mse))


##########################################################################
#########  Case 3. #######################################################
##########################################################################
mse = 10000
obj_loc, distance = generate_data(sensor_loc, k = 7, d = 2, n = 100, original_dist =
 False)
for i in range(100):
    # Your code: compute the mse for this case
    initial_obj_loc = [300,300] + np.random.rand(100,2)
    est_obj_loc = find_mle_by_grad_descent(initial_obj_loc, estimated_sensor_loc, di
stance.T)
    diff = obj_loc - est_obj_loc
```

```
    mse_i = np.linalg.norm(diff)
    mse = min(mse, mse_i)


print('The MSE for Case 2 (if we knew mu is [300,300]) is {}'.format(mse))
```

$$\left(\frac{\partial MSE}{\partial a_i}\right)_l = \sum_k \left(\frac{\partial MSE}{\partial a_{i+1}}\right)_k \quad \sigma'(z_i)_k \cdot (w_i)_{kl}$$

$$= \quad w^T \cdot \left(\sigma'(z_i) \quad \frac{\partial MSE}{\partial a_{i+1}}\right)$$

5 (e)

```
In [108]: run backprop.py
Debugging gradients..
squared difference of layer 0: 2.8053187673930506e-10
squared difference of layer 1: 1.848900869808921e-10
squared difference of layer 2: 1.0137849593546456e-10
```

```
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import pdb

# Gradient descent optimization
# The learning rate is specified by eta
class GDOptimizer(object):
    def __init__(self, eta):
        self.eta = eta

    def initialize(self, layers):
        pass

    # This function performs one gradient descent step
    # layers is a list of dense layers in the network
    # g is a list of gradients going into each layer before the nonlinear activation
    # a is a list of of the activations of each node in the previous layer going
    #
    def update(self, layers, g, a):
        m = a[0].shape[1]
        for layer, curGrad, curA in zip(layers, g, a):
            # TODO: PART F ########################################################
###############
            # Compute the gradients for layer.W and layer.b using the gradient for t
he output of the
            # layer curA and the gradient of the output curGrad
            # Use the gradients to update the weight and the bias for the layer
            #
            # Normalize the learning rate by m (defined above), the number of traini
ng examples input
            # (in parallel) to the network.
            #
            # It may help to think about how you would calculate the update if we in
put just one
            # training example at a time; then compute a mean over these individual
update values.
            # ######################################################################
###############
            pass

# Cost function used to compute prediction errors
class QuadraticCost(object):

    # Compute the squared error between the prediction yp and the observation y
    # This method should compute the cost per element such that the output is the
    # same shape as y and yp
    @staticmethod
    def fx(y,yp):
        # TODO: PART B ########################################################
############
        # Implement me
        # ######################################################################
############
        return 1/2. * ((yp - y) **2)

    # Derivative of the cost function with respect to yp
    @staticmethod
    def dx(y,yp):
        # TODO: PART B ########################################################
############
        # Implement me
        # ######################################################################
############
        return yp - y

# Sigmoid function fully implemented as an example
class SigmoidActivation(object):
    @staticmethod
    def fx(z):
        return 1 / (1 + np.exp(-z))

    @staticmethod
```

```
    def dx(z):
        return SigmoidActivation.fx(z) * (1 - SigmoidActivation.fx(z))

# Hyperbolic tangent function
class TanhActivation(object):

    # Compute tanh for each element in the input z
    @staticmethod
    def fx(z):
        # TODO: PART C ########################################################
############
        # Implement me
        # ####################################################################
############
        return np.tanh(z)

    # Compute the derivative of the tanh function with respect to z
    @staticmethod
    def dx(z):
        # TODO: PART C ########################################################
############
        # Implement me
        # ####################################################################
############
        return 1-np.tanh(z)**2

# Rectified linear unit
class ReLUActivation(object):
    @staticmethod
    def fx(z):
        # TODO: PART C ########################################################
############
        # Implement me
        # ####################################################################
############
        return z * (z>0)

    @staticmethod
    def dx(z):
        # TODO: PART C ########################################################
############
        # Implement me
        # ####################################################################
############
        return 1 * (z>0)

# Linear activation
class LinearActivation(object):
    @staticmethod
    def fx(z):
        # TODO: PART C ########################################################
############
        # Implement me
        # ####################################################################
############
        return z

    @staticmethod
    def dx(z):
        # TODO: PART C ########################################################
############
        # Implement me
        # ####################################################################
############
        return z/z

# This class represents a single hidden or output layer in the neural network
class DenseLayer(object):

    # numNodes: number of hidden units in the layer
    # activation: the activation function to use in this layer
    def __init__(self, numNodes, activation):
        self.numNodes = numNodes
```

```
        self.activation = activation

    def getNumNodes(self):
        return self.numNodes

    # Initialize the weight matrix of this layer based on the size of the matrix W
    def initialize(self, fanIn, scale=1.0):
        s = scale * np.sqrt(6.0 / (self.numNodes + fanIn))
        self.W = np.random.normal(0, s,
                                  (self.numNodes,fanIn))
        self.b = np.random.uniform(-1,1,(self.numNodes,1))

    # Apply the activation function of the layer on the input z
    def a(self, z):
        return self.activation.fx(z)

    # Compute the linear part of the layer
    # The input a is an n x k matrix where n is the number of samples
    # and k is the dimension of the previous layer (or the input to the network)
    def z(self, a):
        return self.W.dot(a) + self.b # Note, this is implemented where we assume a
is k x n

    # Compute the derivative of the layer's activation function with respect to z
    # where z is the output of the above function.
    # This derivative does not contain the derivative of the matrix multiplication
    # in the layer.  That part is computed below in the model class.
    def dx(self, z):
        return self.activation.dx(z)

    # Update the weights of the layer by adding dW to the weights
    def updateWeights(self, dW):
        self.W = self.W + dW

    # Update the bias of the layer by adding db to the bias
    def updateBias(self, db):
        self.b = self.b + db

# This class handles stacking layers together to form the completed neural network
class Model(object):

    # inputSize: the dimension of the inputs that go into the network
    def __init__(self, inputSize):
        self.layers = []
        self.inputSize = inputSize

    # Add a layer to the end of the network
    def addLayer(self, layer):
        self.layers.append(layer)

    # Get the output size of the layer at the given index
    def getLayerSize(self, index):
        if index >= len(self.layers):
            return self.layers[-1].getNumNodes()
        elif index < 0:
            return self.inputSize
        else:
            return self.layers[index].getNumNodes()

    # Initialize the weights of all of the layers in the network and set the cost
    # function to use for optimization
    def initialize(self, cost, initializeLayers=True):
        self.cost = cost
        if initializeLayers:
            for i in range(0,len(self.layers)):
                if i == len(self.layers) - 1:
                    self.layers[i].initialize(self.getLayerSize(i-1))
                else:
                    self.layers[i].initialize(self.getLayerSize(i-1))

    # Compute the output of the network given some input a
    # The matrix a has shape n x k where n is the number of samples and
    # k is the dimension
```

```
    # This function returns
    # yp - the output of the network
    # a - a list of inputs for each layer of the newtork where
    #     a[i] is the input to layer i
    #     (note this does not include the network output!)
    # z - a list of values for each layer after evaluating layer.z(a) but
    #     before evaluating the nonlinear function for the layer
    def evaluate(self, x):
        curA = x.T
        a = [curA]
        z = []
        for layer in self.layers:
            z.append(layer.z(curA))
            curA = layer.a(z[-1])
            a.append(curA)
        yp = a.pop()
        return yp, a, z

    # Compute the output of the network given some input a
    # The matrix a has shape n x k where n is the number of samples and
    # k is the dimension
    def predict(self, a):
        a,_,_ = self.evaluate(a)
        return a.T

    # Computes the gradients at each layer. y is the true labels, yp is the
    # predicted labels, and z is a list of the intermediate values in each
    # layer. Returns the gradients and the forward pass outputs (per layer).
    #
    # In particular, we compute dMSE/dz_i. The reasoning behind this is that
    # in the update function for the optimizer, we do not give it the z values
    # we compute from evaluating the network.
    def compute_grad(self, x, y):
        # Feed forward, computing outputs of each layer and
        # intermediate outputs before the non-linearities
        yp, a, z = self.evaluate(x)

        # d represents (dMSE / da_i) that you derive in part (e);
        #   it is inialized here to be (dMSE / dyp)
        d = self.cost.dx(y.T, yp)
        grad_a = [d]
        grad_z = []

        # Backpropogate the error
        for layer, curZ in zip(reversed(self.layers),reversed(z)):
            # TODO: PART D ############################################################
################
            # Compute the gradient of the output of each layer with respect to the e
rror
            # grad[i] should correspond with the gradient of the output of layer i
            # before the activation is applied (dMSE / dz_i); be sure values are sto
red
            # in the correct ordering!
            # ###########################################################################
################
            grad_z_sigma = np.array(layer.dx(curZ))
            grad_ai = np.dot(layer.W.T, grad_z_sigma * np.array(grad_a[-1]))
            grad_zi = grad_z_sigma * np.array(grad_a[-1])
            grad_a.append(grad_ai)
            grad_z.insert(0, grad_zi)
        return grad_z, a

    # Computes the gradients at each layer. y is the true labels, yp is the
    # predicted labels, and z is a list of the intermediate values in each
    # layer. Uses numerical derivatives to solve rather than symbolic derivatives.
    # Returns the gradients and the forward pass outputs (per layer).
    #
    # In particular, we compute dMSE/dz_i. The reasoning behind this is that
    # in the update function for the optimizer, we do not give it the z values
    # we compute from evaluating the network.
    def numerical_grad(self, x, y, delta=1e-4):

        # computes the loss function output when starting from the ith layer
```

```
        # and inputting z_i
        def compute_cost_from_layer(layer_i, z_i):
            cost = self.layers[layer_i].a(z_i)
            for layer in self.layers[layer_i+1:]:
                cost = layer.a(layer.z(cost))
            return self.cost.fx(y.T, cost)

        # numerically computes the gradient of the error with respect to z_i
        def compute_grad_from_layer(layer_i, inp):
            mask = np.zeros(self.layers[layer_i].b.shape)
            grad_z = []
            # iterate to compute gradient of each variable in z_i, one at a time
            for i in range(mask.shape[0]):
                mask[i] = 1
                delta_p_output = compute_cost_from_layer(layer_i, inp+mask*delta)
                delta_n_output = compute_cost_from_layer(layer_i, inp-mask*delta)
                grad_z.append((delta_p_output - delta_n_output) / (2 * delta))
                mask[i] = 0;

            return np.vstack(grad_z)

        _, a, _ = self.evaluate(x)

        grad = []
        i = 0
        curA = x.T
        for layer in self.layers:
            curA = layer.z(curA)
            grad.append(compute_grad_from_layer(i, curA))
            curA = layer.a(curA)
            i += 1


        return grad, a

    # Train the network given the inputs x and the corresponding observations y
    # The network should be trained for numEpochs iterations using the supplied
    # optimizer
    def train(self, x, y, numEpochs, optimizer):

        # Initialize some stuff
        n = x.shape[0]
        x = x.copy()
        y = y.copy()
        hist = []
        optimizer.initialize(self.layers)

        # Run for the specified number of epochs
        for epoch in range(0,numEpochs):

            # Compute the gradients
            grad, a = self.compute_grad(x, y)

            # Update the network weights
            optimizer.update(self.layers, grad, a)

            # Compute the error at the end of the epoch
            yh = self.predict(x)
            C = self.cost.fx(y, yh)
            C = np.mean(C)
            hist.append(C)
        return hist

if __name__ == '__main__':
    # switch these statements to True to run the code for the corresponding parts
    # PART E
    DEBUG_MODEL = True
    # Part G
    BASE_MODEL = False
    # Part H
    DIFF_SIZES = False
    # Part I
    RIDGE = False
```

```
    # Part J
    SGD = False



    # Generate the training set
    np.random.seed(9001)
    x=np.random.uniform(-np.pi,np.pi,(1000,1))
    y=np.sin(x)
    xLin=np.linspace(-np.pi,np.pi,250).reshape((-1,1))
    yHats = {}

    activations = dict(ReLU=ReLUActivation,
                       tanh=TanhActivation,
                       linear=LinearActivation)
    lr = dict(ReLU=0.02,tanh=0.02,linear=0.005)
    names = ['ReLU','linear','tanh']

    #### PART F ####
    if DEBUG_MODEL:
        print('Debugging gradients..')
        # Build the model
        #activation = activations["linear"]
        activation = activations["ReLU"]
        model = Model(x.shape[1])
        model.addLayer(DenseLayer(10,activation()))
        model.addLayer(DenseLayer(10,activation()))
        model.addLayer(DenseLayer(1,LinearActivation()))
        model.initialize(QuadraticCost())

        grad, _ = model.compute_grad(x, y)
        n_grad, _ = model.numerical_grad(x, y)
        for i in range(len(grad)):
            print('squared difference of layer %d:' % i, np.linalg.norm(grad[i] - n_
grad[i]))


    #### PART G ####
    if BASE_MODEL:
        print('\n----------------------------------------\n')
        print('Standard fully connected network')
        for key in names:
            # Build the model
            activation = activations[key]
            model = Model(x.shape[1])
            model.addLayer(DenseLayer(100,activation()))
            model.addLayer(DenseLayer(100,activation()))
            model.addLayer(DenseLayer(1,LinearActivation()))
            model.initialize(QuadraticCost())

            # Train the model and display the results
            hist = model.train(x,y,500,GDOptimizer(eta=lr[key]))
            yHat = model.predict(x)
            yHats[key] = model.predict(xLin)
            error = np.mean(np.square(yHat - y))/2
            print(key+' MSE: '+str(error))
            plt.plot(hist)
            plt.title(key+' Learning curve')
            plt.show()

        # Plot the approximations
        font = {'family' : 'DejaVu Sans',
            'weight' : 'bold',
                'size'   : 12}
        matplotlib.rc('font', **font)
        y = np.sin(xLin)
        for key in activations:
            plt.plot(xLin,y)
            plt.plot(xLin,yHats[key])
            plt.title(key+' approximation')
            plt.savefig(key+'-approx.png')
            plt.show()
```

```
    # Train with different sized networks
    #### PART H ####
    if DIFF_SIZES:
        print('\n----------------------------------------\n')
        print('Training with various sized network')
        names = ['ReLU', 'tanh']
        sizes = [5,10,25,50]
        widths = [1,2,3]
        errors = {}
        y = np.sin(x)
        for key in names:
            error = []
            for width in widths:
                for size in sizes:
                    activation = activations[key]
                    model = Model(x.shape[1])
                    for _ in range(width):
                        model.addLayer(DenseLayer(size,activation()))
                    model.addLayer(DenseLayer(1,LinearActivation()))
                    model.initialize(QuadraticCost())
                    hist = model.train(x,y,500,GDOptimizer(eta=lr[key]))
                    yHat = model.predict(x)
                    yHats[key] = model.predict(xLin)
                    e = np.mean(np.square(yHat - y))/2
                    error.append(e)
            errors[key] = np.asarray(error).reshape((len(widths),len(sizes)))

        # Print the results
        for key in names:
            error = errors[key]
            print(key+' MSE Error')
            header = '{:^8}'
            for _ in range(len(sizes)):
                header += ' {:^8}'
            headerText = ['Layers'] + [str(s)+' nodes' for s in sizes]
            print(header.format(*headerText))
            for width,row in zip(widths,error):
                text = '{:>8}'
                for _ in range(len(row)):
                    text += ' {:<8}'
                rowText = [str(width)] + ['{0:.5f}'.format(r) for r in row]
                print(text.format(*rowText))

    # Perform ridge regression on the last layer of the network
    #### PART I ####
    if RIDGE:
        print('\n----------------------------------------\n')
        print('Running ridge regression on last layer')
        from sklearn.linear_model import Ridge
        errors = {}
        for key in names:
            error = []
            sizes = [5,10,25,50]
            widths = [1,2,3]
            for width in widths:
                for size in sizes:
                    activation = activations[key]
                    model = Model(x.shape[1])
                    for _ in range(width):
                        model.addLayer(DenseLayer(size,activation()))
                    model.initialize(QuadraticCost())
                    ridge = Ridge(alpha=0.1)
                    X = model.predict(x)
                    ridge.fit(X,y)
                    yHat = ridge.predict(X)
                    e = np.mean(np.square(yHat - y))/2
                    error.append(e)
            errors[key] = np.asarray(error).reshape((len(widths),len(sizes)))

        # Print the results
        for key in names:
            error = errors[key]
            print(key+' MSE Error')
```

```
            header = '{:^8}'
            for _ in range(len(sizes)):
                header += ' {:^8}'
            headerText = ['Layers'] + [str(s)+' nodes' for s in sizes]
            print(header.format(*headerText))
            for width,row in zip(widths,error):
                text = '{:>8}'
                for _ in range(len(row)):
                    text += ' {:<8}'
                rowText = [str(width)] + ['{0:.5f}'.format(r) for r in row]
                print(text.format(*rowText))

        # Plot the results
        for key in names:
            for width,row in zip(widths,error):
                layer = ' layers'
                if width == 1:
                    layer = ' layer'
                plt.semilogy(row,label=str(width)+layer)
            plt.title('MSE for ridge regression with '+key+' activation')
            plt.xticks(range(len(sizes)),sizes)
            plt.xlabel('Layer size')
            plt.ylabel('MSE')
            plt.legend()
            plt.savefig(key+'-ridge.png')
            plt.show()

    #### BONUS PART J ####
    if SGD:
        # Test for SGD... Implement!
        pass
```

6. Your own question

Q: prove that the evecs of a symmetric operator are orthogonal to each other

A: $A\vec{x} = \alpha\vec{x}$

$\vec{x}^T A^T = \alpha\vec{x}^T$

Since A is symmetric, $A = A^T$

$\vec{x}^T A = \alpha\vec{x}^T$

Similarly for $\vec{y}$, $A\vec{y} = \beta\vec{y}$

$\vec{x}^T A\vec{y} = \alpha\vec{x}^T\vec{y} = \beta\vec{x}^T\vec{y}$

so $(\alpha-\beta)\vec{x}^T\vec{y} = 0$

$\vec{x}^T\vec{y} = 0$

so $\vec{x}$ and $\vec{y}$ are orthogonal to each other