CS 189: Introduction to Machine Learning

Fall 2017

•

Homework 6

•

Solutions by

Jinhong Du

3033483677

Question 1

(a)

Jinhong Du
jaydu@berkeley.edu

(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. Jinhong Du

Question 2

(a)

$\because \quad \forall\ x_1, x_2 \in \mathrm{R}^d,\ \forall\ \lambda \in \mathrm{R},\ \lambda \in [0,1],$

$$
\begin{aligned}
f[\lambda x_1 + (1-\lambda)x_2] &= \|\lambda x_1 + (1-\lambda)x_2 - b\|_2 \\
&= \|\lambda(x_1 - b) + (1-\lambda)(x_2 - b)\|_2 \\
&\leqslant \|\lambda x_1\|_2 + \|(1-\lambda)x_2 - b\|_2 \\
&= |\lambda|\|x_1 + (1-\lambda)x_2 - b\|_2 + |1-\lambda|\|x_2 - b\|_2 \\
&= \lambda f(x_1) + (1-\lambda)f(x_2)
\end{aligned}
$$

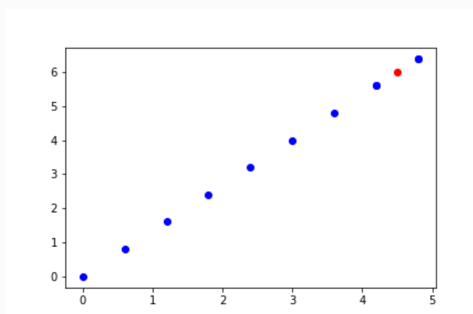
$\therefore \quad f(x)$ is a convex function of x

(b)

$$
\begin{aligned}
f(x) &= \|x - b\|_2 \\
&= \sqrt{(x_1 - 4.5)^2 + (x_2 - 6)^2} \geqslant 0
\end{aligned}
$$

the equation holds when $x^* = \begin{pmatrix} 4.5 & 6 \end{pmatrix}^T$

```
[[ 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]]
```



It didn't find the optimal solution. It is because that a step size is too big and x_i moves left and right of the optimal point $\begin{pmatrix} 4.5 & 6 \end{pmatrix}^T$. At the end, x_i change from either $\begin{pmatrix} 4.2 & 5.6 \end{pmatrix}^T$ or $\begin{pmatrix} 4.8 & 6.4 \end{pmatrix}^T$ to the other.

To general $b \neq \vec{0}$, it depends on the choice of b. For example, if $b = \begin{pmatrix} 0.6 & 0.8 \end{pmatrix}^T$, then we can get the optimal solution at the first step. However, if we choose $b = \begin{pmatrix} 4.5 & 6 \end{pmatrix}^T$, it will fail.

$\because$

$$\nabla f(x) = \frac{x - b}{\|x - b\|_2}$$

$\therefore$

$$x_{i+1} = x_i - \frac{x_i - b}{\|x_i - b\|_2}$$

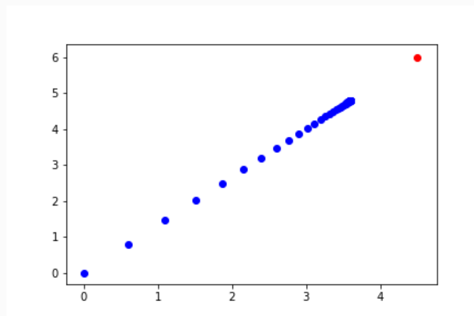
Solution (cont.)

We know that $\dfrac{x_i - b}{\|x_i - b\|_2}$ is a unit vector with the direction $\pm\overrightarrow{Ox^*} = \pm(4.5, 6) = \pm(0.6, 0.8)$. And $4.5 = 0.6 \times 7 + 0.3 = 0.6 \times 8 - 0.3,\ 6.0 = 0.8 \times 7 + 0.4 = 0.8 \times 8 - 0.4$, so the optimal solution will switch between $(4.2, 5.6)$ and $(4.8, 6.4)$.

(c)

After running for 10000 steps, the optimal solution of gradient descend approximates the point $\begin{pmatrix} 3.6 & 4.8 \end{pmatrix}^T$.

```
[[ 0.          0.        ]
 [ 0.6         0.8       ]
 [ 1.1         1.46666667]
 ...,
 [ 3.6         4.8       ]
 [ 3.6         4.8       ]
 [ 3.6         4.8       ]]
```



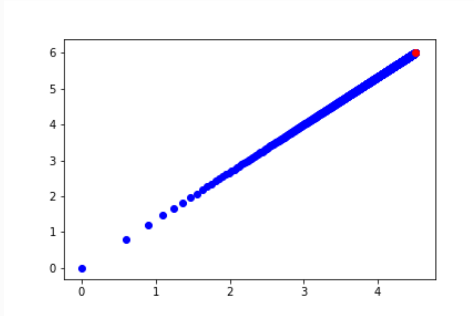
The gradient descend cannot find the optimal solution because the step size is getting smaller and smaller such that it moves slowly from x_i to x_{i+1} when i is large.
Suppose that $\|x_i\|_2 < \left\| \begin{pmatrix} 4.5 \\ 6 \end{pmatrix} \right\|_2$,

$$\begin{aligned} x_{i+1} &= x_i - \left(\frac{5}{6}\right)^i \frac{x_i - b}{\|x_i - b\|_2} \\ &= x_i + \left(\frac{5}{6}\right)^i \begin{pmatrix} 0.6 \\ 0.8 \end{pmatrix} \\ &= x_0 + \sum_{j=0}^{i} \left(\frac{5}{6}\right)^j \begin{pmatrix} 0.6 \\ 0.8 \end{pmatrix} \\ \lim_{i \to \infty} x_{i+1} &= \sum_{j=0}^{\infty} \left(\frac{5}{6}\right)^j \begin{pmatrix} 0.6 \\ 0.8 \end{pmatrix} \\ &= \begin{pmatrix} 3.6 \\ 4.8 \end{pmatrix} \end{aligned}$$

$$\left\| \begin{pmatrix} 3.6 \\ 4.8 \end{pmatrix} \right\|_2 < \left\| \begin{pmatrix} 4.5 \\ 6 \end{pmatrix} \right\|_2$$

so the assumption holds.

(d)



The gradient descend find the optimal solution. At x_{1005} it will get within 0.01 of the optimal solution. Suppose that $\|x_i\|_2 < \left\|\begin{pmatrix}4.5\\6\end{pmatrix}\right\|_2$,

$$
\begin{aligned}
x_{i+1} &= x_i - \left(\frac{1}{i+1}\right)^i \frac{x_i - b}{\|x_i - b\|_2} \\
&= x_i + \left(\frac{1}{i+1}\right)^i \begin{pmatrix}0.6\\0.8\end{pmatrix} \\
&= x_0 + \sum_{j=0}^{i} \left(\frac{1}{i+1}\right)^j \begin{pmatrix}0.6\\0.8\end{pmatrix} \\
\lim_{i\to\infty} x_{i+1} &= \sum_{j=0}^{\infty} \left(\frac{1}{j+1}\right)^j \begin{pmatrix}0.6\\0.8\end{pmatrix} \\
&= \begin{pmatrix}\infty\\\infty\end{pmatrix}
\end{aligned}
$$

$$
\left\|\begin{pmatrix}\infty\\\infty\end{pmatrix}\right\|_2 > \left\|\begin{pmatrix}4.5\\6\end{pmatrix}\right\|_2
$$

so the assumption don't holds. So $\exists\ n_0 \in \mathbb{N}$, s.t. $\left\|\sum_{j=0}^{n_0} \left(\frac{1}{j+1}\right)^j \begin{pmatrix}0.6\\0.8\end{pmatrix}\right\|_2 \leqslant \left\|\begin{pmatrix}4.5\\6\end{pmatrix}\right\|_2$ and $\left\|\sum_{j=0}^{n_0+1} \left(\frac{1}{j+1}\right)^j \begin{pmatrix}0.6\\0.8\end{pmatrix}\right\|_2 > \left\|\begin{pmatrix}4.5\\6\end{pmatrix}\right\|_2$. If the equation holds, then we find the optimal solution at the n_0th step. Otherwise, $\exists\ n_1 > n_0$, s.t.

$$
\left\|\left[\sum_{j=0}^{n_0} \left(\frac{1}{j+1}\right)^j - \sum_{j=n_0+1}^{n_1} \left(\frac{1}{j+1}\right)^j + \cdots\right] \begin{pmatrix}0.6\\0.8\end{pmatrix}\right\|_2 \geqslant \left\|\begin{pmatrix}4.5\\6\end{pmatrix}\right\|_2
$$

and

$$
\left\|\left[\sum_{j=0}^{n_0} \left(\frac{1}{j+1}\right)^j - \sum_{j=n_0+1}^{n_1+1} \left(\frac{1}{j+1}\right)^j + \cdots\right] \begin{pmatrix}0.6\\0.8\end{pmatrix}\right\|_2 < \left\|\begin{pmatrix}4.5\\6\end{pmatrix}\right\|_2
$$

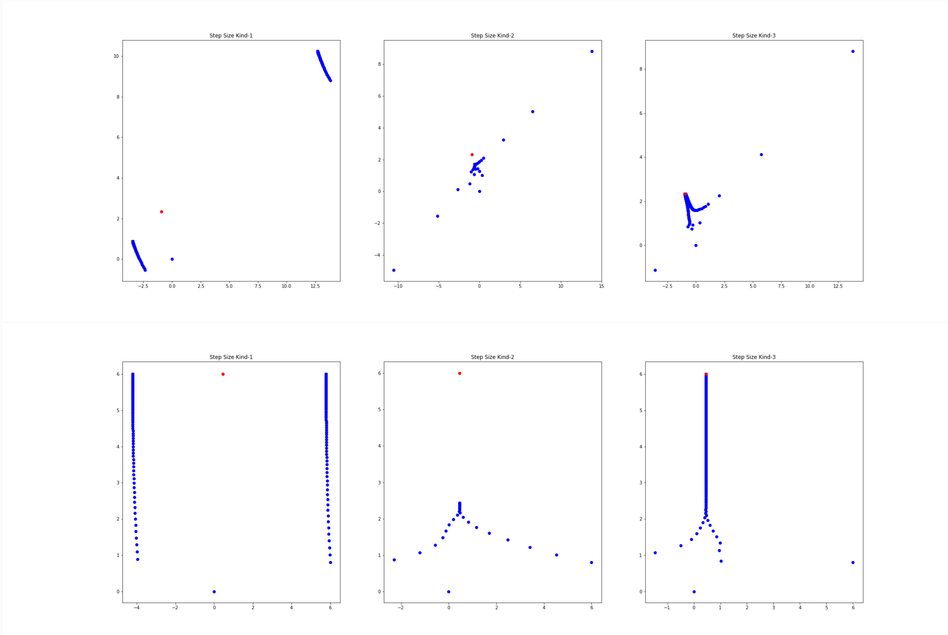
Repeat there precedures, we can get that the sequence $\{n_0, n_1, \cdots\}$

$$
\left\|\left[\sum_{j=0}^{n_0} \left(\frac{1}{j+1}\right)^j - \sum_{j=n_0+1}^{n_1} \left(\frac{1}{j+1}\right)^j + \cdots\right] \begin{pmatrix}0.6\\0.8\end{pmatrix}\right\|_2 \to \left\|\begin{pmatrix}4.5\\6\end{pmatrix}\right\|_2
$$

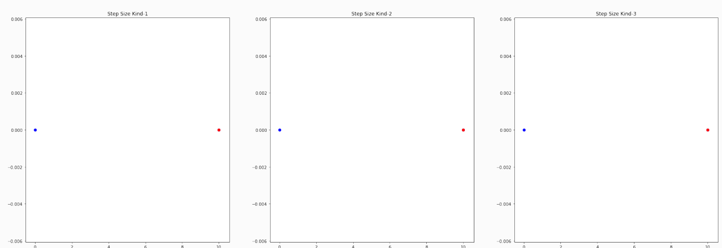
since the remainder of series is infinite. Or in one of n_i, we approach the optimal solution.

(e)

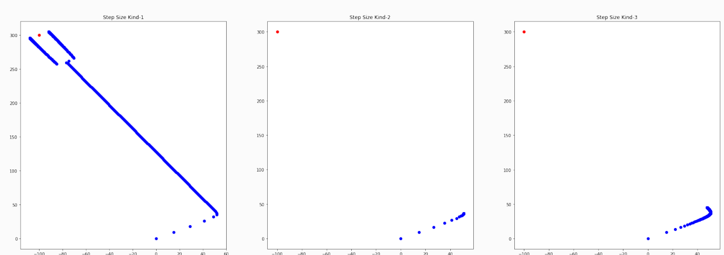
For the given A, only the 3th kind of steps choices finds the optimal solution.



For different choices of b, the results are different. For the first choice of A and $b = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$, all kinds of steps choices work.



For the second choice of A and $b = \begin{pmatrix} 900 \\ 900 \end{pmatrix}$, all kinds of steps choices don't work.



Question 3

(a)

$\because$

$$\begin{aligned}\nabla f(x) &= \frac{1}{2}\nabla\|Ax-b\|_2^2\\ &= \frac{1}{2}\nabla(Ax-b)^T(Ax-b)\\ &= \frac{1}{2}\nabla[x^TA^TAx - b^TAx - x^TA^Tb + b^Tb]\\ &= A^TAx - A^Tb\end{aligned}$$

given $b = \vec{0}$,

$$\nabla f(x) = A^TAx$$

$\therefore$

$$\begin{aligned}x_{i+1} &= x_i - \gamma\nabla f(x_i)\\ &= x_i - \gamma A^TAx_i\\ &= (I - \gamma A^TA)x_i\end{aligned}$$

with the initial condition x_0.

(b)

$\because$

$$x_i = (I - \gamma A^TA)^i x_0$$

We want $\forall\ x_0$, and $\epsilon_0 = x_0 - x$ where x is the optimal solution we are looking for, the following condition is satisfied,

$$\begin{aligned}\epsilon_k &= x_k - x\\ &= (I - \gamma A^TA)\epsilon_{k-1}\\ &= (I - \gamma A^TA)^k\epsilon_0 \to 0\end{aligned}$$

$\therefore$

$$\lim_{k\to\infty}(I - \gamma A^TA)^k = 0$$

i.e. x_i won't blow up arbitrarily over time.

(c)

$\because$

$$\begin{aligned}\varphi(x) &= x - \gamma\nabla f(x)\\ &= x - \gamma(A^TAx - A^Tb)\\ &= (I - \gamma A^TA)x + \gamma(A^Tb)\end{aligned}$$

Solution (cont.)

and is a $d \times d$ symmetric matrix, where $a_i \in \mathbb{R}^d$

$\therefore$

$$I - \gamma A^T A = P^{-1} J P$$

where P is ao orthonormal matrix and $J = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ & 0 & \cdots & \lambda_d \end{pmatrix}$ is a diagonal matrix

$\therefore$

$$\begin{aligned} \|\varphi(x) - \varphi(x')\|_2 &= \|(I - \gamma A^T A)(x - x')\|_2 \\ &= \|P^T J P(x - x^*)\|_2 \\ &= \sqrt{(x - x^*)^T P^T J P P^T J P (x - x^*)} \\ &= \sqrt{(x - x^*)^T P^T J^2 P (x - x^*)} \\ &\xlongequal{y = P(x - x^*)} \sqrt{y^T J^2 y} \\ &= \sqrt{\sum_{i=1}^{n} \lambda_i^2 y_i^2} \\ &\leqslant \max_i\{|\lambda_i|\} \sqrt{\sum_{i=1}^{n} y_i^2} \\ &= |\lambda_{\max}(I - \gamma A^T A)| \|y\|_2 \end{aligned}$$

$\because$

$$\lambda_{\min}(I - \gamma A^T A) \leqslant \lambda_{\max}(I - \gamma A^T A) \leqslant \lambda_{\max}(I - \gamma A^T A)$$

$\therefore$

$$|\lambda_{\max}(I - \gamma A^T A)| \leqslant \max\{|\lambda_{\min}(I - \gamma A^T A)|, |\lambda_{\max}(I - \gamma A^T A)|\}$$

$\therefore$

$$\|\varphi(x) - \varphi(x')\|_2 \leqslant \beta \|x - x'\|_2$$

(d)

$\because$

$$x^* = \arg\min_{x \in \mathbb{R}^d} f(x)$$

$\therefore$

$$\nabla f(x^*) = 0$$

$\therefore$

$$\begin{aligned} \|\varphi(x_k) - \varphi(x^*)\|_2 &= \|x_k - \gamma(A^T A x_k - A^T b) - x^*\|_2 \\ &= \|(I - \gamma A^T A) x_k - x^*\|_2 \\ &= \|x_{k+1} - x^*\|_2 \end{aligned}$$

Solution (cont.)

$\because$

$$\begin{aligned}\|x_{k+1}-x^*\|_2 &= \|\varphi(x_k)-\varphi(x^*)\|_2 \\ &\leqslant \beta\|x_k-x^*\|_2\end{aligned}$$

$\therefore$

$$\|x_{k+1}-x^*\|_2 \leqslant \beta^{k+1}\|x_0-x^*\|_2$$

(e)

$\because$ x^* is the minimizer and

$$\nabla f(x) = A^T Ax - A^T b$$

$\therefore$ x^* equals to the solution to $\nabla f(x) = 0$, i.e. $A^T Ax^* = A^T b$

i.e.

$$x^* = (A^T A)^{-1} A^T b$$

$\therefore$

$$\begin{aligned}f(x^*) &= \frac{1}{2}\|Ax^*-b\|_2^2 \\ &= \frac{1}{2}(Ax^*-b)^T(Ax^*-b) \\ &= \frac{1}{2}[x^{*T}A^TAx^* - b^TAx^* - x^{*T}A^Tb + b^Tb] \\ &= \frac{1}{2}[b^TA(A^TA)^{-1}(A^TA)(A^TA)^{-1}A^Tb - 2b^TA(A^TA)^{-1}A^Tb + b^Tb] \\ &= \frac{1}{2}[-b^TA(A^TA)^{-1}A^Tb + b^Tb] \\ &= \frac{1}{2}[-b^TA(A^TA)^{-1}(A^TA)(A^TA)^{-1}A^Tb + b^Tb] \\ &= \frac{1}{2}(-x^{*T}A^TAx^* + b^Tb)\end{aligned}$$

$\therefore$

$$\begin{aligned}f(x) &= \frac{1}{2}\|Ax-b\|_2^2 \\ &= \frac{1}{2}[(Ax-b)^T(Ax-b)] \\ &= \frac{1}{2}[x^TA^TAx - b^TAx - x^TA^Tb + b^Tb] \\ &= \frac{1}{2}[(x-x^*)^TA^TA(x-x^*) - b^TAx - x^TA^Tb + x^TA^TAx^* + x^{*T}A^TAx - x^{*T}A^TAx^* + b^Tb] \\ &= \frac{1}{2}[\|A(x-x^*)\|_2^2 - b^TAx - x^TA^Tb + x^TA^Tb + bA^TAx - x^{*T}A^TAx^* + b^Tb] \\ &= \frac{1}{2}[\|A(x-x^*)\|_2^2 - x^{*T}A^TAx^*] \\ &= \frac{1}{2}\|A(x-x^*)\|_2^2 - f(x^*)\end{aligned}$$

$\therefore$

$$f(x) - f(x^*) = \frac{1}{2}\|A(x-x^*)\|_2^2$$

(f)

$\because$ A^TA is a symmetric matrix

$\therefore$ $\exists\ P, J \in \mathbb{R}^{d\times d}$ where P is orthonormal, $P^T = P^{-1}$ and $J = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \lambda_2 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \cdots & \lambda_d \end{pmatrix}$

$\therefore$

$$\begin{aligned}
\|A(x-x^*)\|_2^2 &= (x-x^*)^TA^TA(x-x^*) \\
&= (x-x^*)^TP^TJP(x-x^*) \\
&\xlongequal{y=P(x-x^*)} y^TJy \\
&= \sum_{i=1}^{d} \lambda_i y_i^2 \\
&\leqslant \max_i\{\lambda_i\}\|y\|_2^2 \\
&= \max_i\{\lambda_i\}\|P(x-x^*)\|_2^2 \\
&= \lambda_{\max}(A^TA)\|x-x^*\|_2^2 \\
&= \alpha\|x-x^*\|_2^2
\end{aligned}$$

$\therefore$

$$f(x_k) - f(x^*) \leqslant \frac{\alpha}{2}\|x_k - x^*\|_2^2$$

From (e) we have

$$\begin{aligned}
f(x_k) - f(x^*) &\leqslant \frac{\alpha}{2}(\beta^k\|x_k - x^*\|_2)^2 \\
&= \frac{\alpha}{2}\beta^{2k}\|x_k - x^*\|_2^2
\end{aligned}$$

(g)

$\because$ A^TA is positive definite

$\therefore$

$$\lambda_{\max}(A^TA) \geqslant \lambda_{\min}(A^TA) > 0$$

$\because$

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

Solution (cont.)

$\therefore \quad \forall\, \epsilon > 0$, $\exists\, n \in \mathbb{N}$, s.t. $0 < \left(\dfrac{\lambda_{\min}(A^TA)}{\lambda_{\max}(A^TA)}\right)^n < \epsilon$, let $\gamma = \dfrac{1 - \left(\frac{\lambda_{\min}(A^TA)}{\lambda_{\max}(A^TA)}\right)^n}{\lambda_{\min}(A^TA)}$, we have

$$
\begin{aligned}
1 - \gamma\lambda_{\min}(A^TA) &= \left(\frac{\lambda_{\min}(A^TA)}{\lambda_{\max}(A^TA)}\right)^n < \epsilon \\
1 - \gamma\lambda_{\min}(A^TA) &\geqslant 1 - \gamma\lambda_{\max}(A^TA) \\
&= 1 - \left[1 - \left(\frac{\lambda_{\min}(A^TA)}{\lambda_{\max}(A^TA)}\right)^n\right]\frac{\lambda_{\max}(A^TA)}{\lambda_{\min}(A^TA)} \\
&\geqslant 1 - \left(1 + \frac{\lambda_{\min}(A^TA)}{\lambda_{\max}(A^TA)}\right)\frac{\lambda_{\max}(A^TA)}{\lambda_{\min}(A^TA)} \\
&> -\epsilon \\
1 - \gamma\lambda_{\max}(A^TA) &= 1 - \left[1 - \left(\frac{\lambda_{\min}(A^TA)}{\lambda_{\max}(A^TA)}\right)^n\right]\frac{\lambda_{\max}(A^TA)}{\lambda_{\min}(A^TA)} \\
&\leqslant 1 - \left[1 - \left(\frac{\lambda_{\min}(A^TA)}{\lambda_{\max}(A^TA)}\right)^n\right] \\
&= \left(\frac{\lambda_{\min}(A^TA)}{\lambda_{\max}(A^TA)}\right)^n < \epsilon
\end{aligned}
$$

i.e.

$$\beta < \epsilon$$

When picking $\gamma = \dfrac{1 - \left(\frac{\lambda_{\min}(A^TA)}{\lambda_{\max}(A^TA)}\right)^n}{\lambda_{\min}(A^TA)}$, we have

$$
\begin{aligned}
f(x) - f(x^*) &\leqslant \frac{\alpha}{2}\beta^{2k}\|x_0 - x^*\|_2^2 \\
&= \frac{\alpha}{2}\left(\frac{\lambda_{\min}(A^TA)}{\lambda_{\max}(A^TA)}\right)^{2nk}\|x_0 - x^*\|_2^2 \\
&= \frac{\alpha}{2}Q^{2nk}\|x_0 - x^*\|_2^2
\end{aligned}
$$

Question 4

(a)

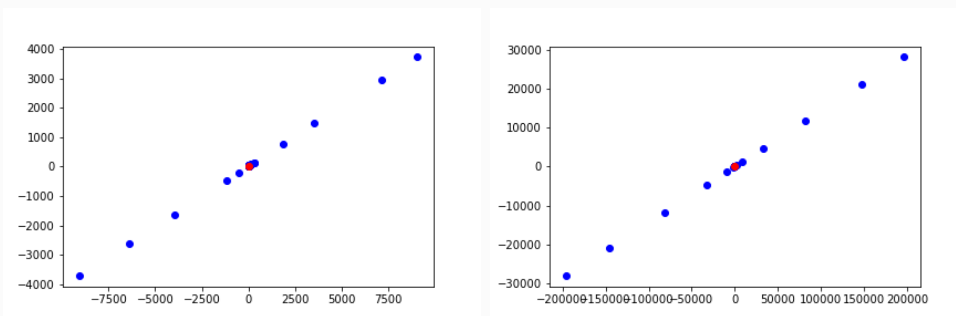
$$L(x_1, y_1) = -\sum_{i=1}^{7}(\sqrt{(a_i - x_1)^2 + (b_i - y_1)^2} - d_i)^2$$

From the chain rule,

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

$$\frac{\partial L}{\partial x_2} = 2\sum_{i=1}^{7}(\sqrt{(a_i - x_1)^2 + (b_i - y_1)^2} - d_i)\frac{(b_i - y_1)}{\sqrt{(a_i - x_1)^2 + (b_i - y_1)^2}}$$

(b)



The optimal solutions of these two approaches are

$$\begin{pmatrix}43.07188566 & 32.71217991\end{pmatrix} \qquad \begin{pmatrix}43.0718848 & 32.7121788\end{pmatrix}$$

Starting at a random point seems not as stable as the the former. Sometimes it takes less time to find the optimal solution while sometimes it will take longer time. I choose steps $\gamma_i = \frac{1}{i+1}$

```
def gradient(obj_loc, d, sen_loc):
    g = 2*np.dot(((np.linalg.norm(obj_loc-sen_loc, axis=1)-d
        )/np.linalg.norm(obj_loc-sen_loc, axis=1)).T,(obj_loc-sen_loc))
    if np.any(np.isnan(g)):
        return 0
    else:
        return g

def gradient_descend_step(obj_loc, d, sen_loc, step_count,
        step_size):
    obj_loc = obj_loc - step_size(step_count) * gradient(obj_loc,
        d, sen_loc)
    return np.array(obj_loc)
```

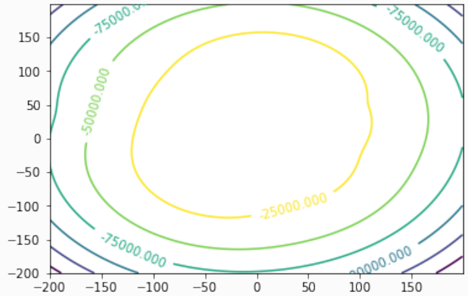
Solution (cont.)

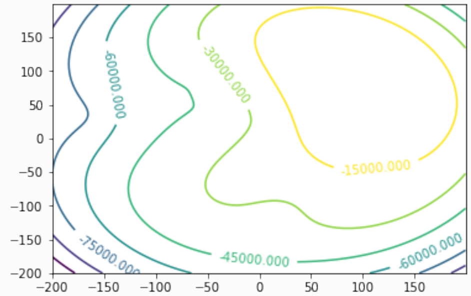
```python

def gradient_descend(obj_loc, d, sen_loc, total_step_count,
        step_size, err=0):
    positions = [np.array(obj_loc)]
    for k in range(total_step_count):
        positions.append(gradient_descend_step(positions[-1],
            d, sen_loc, k, step_size))

    return np.array(positions)

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
err = 1e-15
positions = gradient_descend(initial_position, single_distance,
        sensor_loc, total_step_count, step_size, err)
print(positions)
plt.scatter(positions[:,0], positions[:,1], c='blue')
plt.scatter(obj_loc[:,0], obj_loc[:,1], c='red')
plt.plot()
plt.savefig('4b1.png')
plt.show()

initial_position = 100*np.random.randn(2)
print(initial_position)
total_step_count = 1000  # number of GD steps to take
step_size = lambda i: 1/(i+1)  # step size at iteration i
err = 1e-15
positions = gradient_descend(initial_position, single_distance,
      sensor_loc, total_step_count, step_size, err)
print(positions)
plt.scatter(positions[:,0], positions[:,1], c='blue')
plt.scatter(obj_loc[:,0], obj_loc[:,1], c='red')
plt.plot()
plt.savefig('4b2.png')
plt.show()
```

(c)

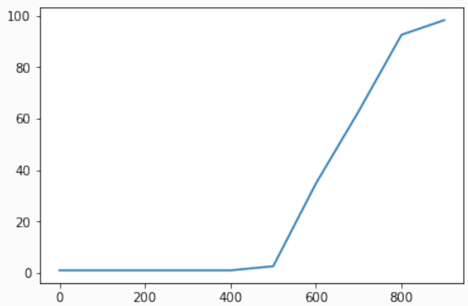
Contour plot for $x_1 = (0, 0)$ and $x_1 = (100, 100)$:

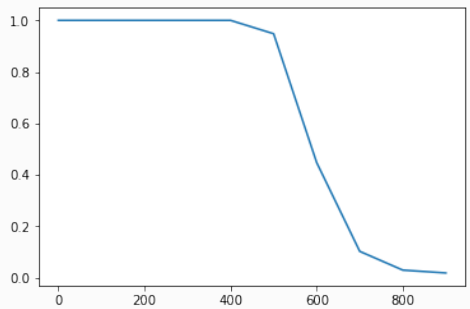
Solution (cont.)





The average number of local minima against x_1:
The average proportion against x_1:





When the $\|x_i\|_2$ is big enough, the easier to find the local minima. However, it will find more local minima than the global minima.

```
def gradient(obj_loc, d, sen_loc):
    g = 2*np.dot(((np.linalg.norm(obj_loc-sen_loc,axis=1)-d)
      /np.linalg.norm(obj_loc-sen_loc,axis=1)).T,(obj_loc-sen_loc))
    if np.any(np.isnan(g)):
        return 0
    else:
        return g

def gradient_descend_step(obj_loc, d, sen_loc, step_count,
      step_size):
    obj_loc = obj_loc - step_size(step_count) * gradient(obj_loc, d,
        sen_loc)
    return np.array(obj_loc)

def gradient_descend(obj_loc, d, sen_loc, total_step_count,
      step_size, err=0):
    positions = np.array(obj_loc)
    for k in range(total_step_count):
        positions = gradient_descend_step(positions, d,
            sen_loc, k, step_size)

```

Solution (cont.)

```python
    return positions

def generate_data_given_location(sensor_loc, obj_loc, k = 7,
      d = 2, sigma=1):
    assert k, d == sensor_loc.shape

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

def log_likelihood(obj_loc, sensor_loc, distance):
    diff_distance = np.sqrt(np.sum((sensor_loc - obj_loc)**2,
      axis = 1))- distance
    func_value = -sum((diff_distance)**2)/2
    return func_value

np.random.seed(100)
# Sensor locations.
sensor_loc = generate_sensors()
num_gd_replicates = 100

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

distances = []
for i in range(len(obj_locs)):
      distances.append(generate_data_given_location(sensor_loc,
           obj_locs[i])[0])
distances = np.array(distances)

x_axis = np.arange(-200.0, 200.0, 1)
y_axis = np.arange(-200.0, 200.0, 1)
X, Y = np.meshgrid(x_axis, y_axis)
m ,n = np.shape(X)
like = [[log_likelihood([X[j][i],Y[j][i]],sensor_loc,
      distances[0][0]) for i in range(n)] for j in range(m)]
CS = plt.contour(X, Y, like)
plt.clabel(CS, inline=1, fontsize=10)
plt.savefig('4c1.png')
plt.show()
```

Solution (cont.)

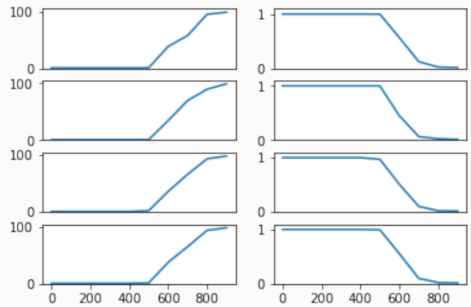
```
x_axis = np.arange(-200.0, 200.0, 1)
y_axis = np.arange(-200.0, 200.0, 1)
X, Y = np.meshgrid(x_axis, y_axis)
m ,n = np.shape(X)
like = [[log_likelihood([X[j][i],Y[j][i]],sensor_loc,distances[1][0])
  for i in range(n)] for j in range(m)]
CS = plt.contour(X, Y, like)
plt.clabel(CS, inline=1, fontsize=10)
plt.savefig('4c2.png')
plt.show()

total_step_count = 1000  # number of GD steps to take
step_size = lambda i: 0.1  # step size at iteration i
err = 0
positions = []
for i in range(len(obj_locs)):
    position = []
    for j in range(10):
        optimal_solution = np.zeros((num_gd_replicates,2))
        initial_position = np.random.randn(num_gd_replicates,2)*(
            obj_locs[i][0][0]+1)
        for k in range(num_gd_replicates):
            optimal_solution[k,:] = gradient_descend(
                initial_position[k], distances[i], sensor_loc,
                  total_step_count, step_size, err)[-1]
        position += [optimal_solution]
    print(i)
    positions += [np.array(position)]
positions = np.array(positions)

result = np.zeros((10,10))
for i in range(10):
    for j in range(10):
        result[i,j] = len(np.unique(np.round(positions[i][j],2),
          axis=0))
average_result = np.mean(result,axis=1)
plt.plot(np.arange(0,1000,100),average_result)

result2 = np.zeros((10,10))
for i in range(10):
    for j in range(10):
        result2[i,j] = max(np.unique(np.round(positions[i][j],2),
          axis=0,return_counts=True)[1])
```

Solution (cont.)

```
average_result2 = np.mean(result2, axis=1)/ num_gd_replicates
plt.plot(np.arange(0,1000,100), average_result2)
```

(d)

We can see that the average number of local minima against x_1 and the average proportion against x_1 remain almost the same when change the variance of the measurement noise.



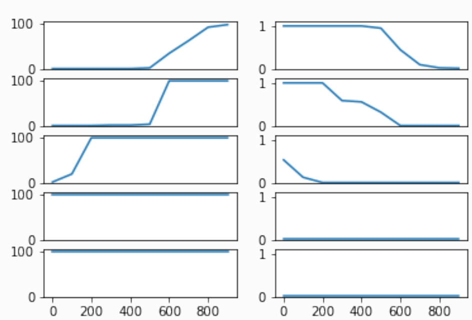

```
np.random.seed(100)
sensor_loc = generate_sensors()
num_gd_replicates = 100
sigma = np.arange(1,5)

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

total_step_count = 1000  # number of GD steps to take
step_size = lambda i: 0.1  # step size at iteration i
err = 0
positions2 = []

for e in range(len(sigma)):
    distance2 = []
    for i in range(len(obj_locs)):
        distance2.append(generate_data_given_location(sensor_loc,
            obj_locs[i], sigma=10^(-i))[0])
    distance2 = np.array(distance2)

    position2 = []
    for i in range(len(obj_locs)):
        p = []
        for j in range(10):
            optimal_solution2 = np.zeros((num_gd_replicates,2))
            initial_position2 = np.random.randn(num_gd_replicates,2)
              *(obj_locs[i][0][0]+1)
```

Solution (cont.)

```
                for k in range(num_gd_replicates):
                    optimal_solution2[k] = gradient_descend(
                      initial_position2[k], distance2[i], sensor_loc,
                      total_step_count, step_size, err)[-1]
                p += [optimal_solution2]
            position2 += [np.array(p)]
        position2 = np.array(position2)
        positions2.append(position2)

        plt.subplot(len(sigma),2,2*e+1)
        result21 = np.zeros((10,10))
        for i in range(10):
            for j in range(10):
                result21[i,j] = len(np.unique(np.round(position2[i][j],
                    2),axis=0))
        average_result21 = np.mean(result21,axis=1)
        plt.plot(np.arange(0,1000,100),average_result21)
        plt.subplot(len(sigma),2,2*e+2)
        result22 = np.zeros((10,10))
        for i in range(10):
            for j in range(10):
                result22[i,j] = max(np.unique(np.round(position2[i][j],
                    2),axis=0,return_counts=True)[1])
        average_result22 = np.mean(result22,axis=1)/ num_gd_replicates
        plt.plot(np.arange(0,1000,100),average_result22)
        print(e)
plt.show()
```

(e)

We can see that the average number of local minima against x_1 increases and the average proportion against x_1 decreases when number of sensor increases. I.e, when increasing the number of sensors, it is much easier to find the local minima but harder to find the global minima.



Solution (cont.)

```
np.random.seed(100)
# Sensor locations.
num_gd_replicates = 100
num_sensor = np.arange(7,28,5)

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

total_step_count = 1000  # number of GD steps to take
step_size = lambda i: 0.1  # step size at iteration i
err = 0
positions3 = []

for e in range(len(num_sensor)):
    sensor_loc = generate_sensors(k=num_sensor[e])
    distance3 = []
    for i in range(len(obj_locs)):
        distance3.append(generate_data_given_location(sensor_loc,
            obj_locs[i], k=num_sensor[e])[0])
    distance3 = np.array(distance3)

    position3 = []
    for i in range(len(obj_locs)):
        p = []
        for j in range(10):
            optimal_solution3 = np.zeros((num_gd_replicates,2))
            initial_position3 = np.random.randn(num_gd_replicates,2)
              *(obj_locs[i][0][0]+1)
            for k in range(num_gd_replicates):
                optimal_solution3[k] = gradient_descend(
                    initial_position3[k], distance3[i], sensor_loc,
                      total_step_count, step_size, err)
            p += [optimal_solution3]
        position3 += [np.array(p)]
    position3 = np.array(position3)
    positions3.append(position3)

    plt.subplot(len(num_sensor),2,2*e+1)
    result31 = np.zeros((10,10))
    for i in range(10):
        for j in range(10):
            result31[i,j] = len(np.unique(np.round(position3[i][j],2
                ),axis=0))
```

Solution (cont.)

```
        average_result31 = np.mean(result31, axis=1)
        plt.plot(np.arange(0,1000,100), average_result31)
        plt.subplot(len(num_sensor),2,2*e+2)
        result32 = np.zeros((10,10))
        for i in range(10):
            for j in range(10):
                result32[i,j] = max(np.unique(np.round(position3[i][j],2
                    ),axis=0,return_counts=True)[1])
        average_result32 = np.mean(result32, axis=1)/ num_gd_replicates
        plt.plot(np.arange(0,1000,100), average_result32)
        print(e)
np.save('4e', positions3)
plt.show()
```

(f)

$$L(a_1,\cdots,a_6,b_1,\cdots,b_7) = -\sum_{i=1}^{7}\sum_{j=1}^{100}(\sqrt{(a_i-x_j)^2+(b_i-y_j)^2}-d_{ij})^2$$

From the chain rule,

$$\frac{\partial L}{\partial a_i} = -2\sum_{j=1}^{100}(\sqrt{(a_i-x_1)^2+(b_i-y_1)^2}-d_{ij})\frac{(a_i-x_j)}{\sqrt{(a_i-x_j)^2+(b_i-y_j)^2}}$$

$$\frac{\partial L}{\partial b_i} = -2\sum_{j=1}^{100}(\sqrt{(a_i-x_1)^2+(b_i-y_1)^2}-d_{ij})\frac{(b_i-x_j)}{\sqrt{(a_i-x_j)^2+(b_i-y_j)^2}}$$

DataSet	Train	Test1	Test2
MSE	1469.01800279	10163.56297	657407.238113

```
np.random.seed(100)
# Sensor locations.
sensor_loc = generate_sensors()

train = np.random.randn(100,2)*100
test1 = np.random.randn(100,2)*100
test2 = 300+np.random.randn(100,2)*100

train_distances = []
for i in range(len(train)):
    train_distances.append(generate_data_given_location(sensor_loc,
        train[i,np.newaxis])[0])
train_distances = np.array(train_distances)
```

Solution (cont.)

```python
test1_distances = []
for i in range(len(test1)):
    test1_distances.append(generate_data_given_location(sensor_loc,
        test1[i,np.newaxis])[0])
test1_distances = np.array(test1_distances)
test2_distances = []
for i in range(len(test2)):
    test2_distances.append(generate_data_given_location(sensor_loc,
        test2[i,np.newaxis])[0])
test2_distances = np.array(test2_distances)

def gradient1(obj_loc, d, sen_loc):
    g = np.zeros_like(sen_loc)
    for i in range(len(g)):
        g[i] = -2*np.dot(((np.linalg.norm(obj_loc-sen_loc[i],axis=1)
          -d[:,i])/np.linalg.norm(obj_loc-sen_loc[i],axis=1)).T,
          (obj_loc-sen_loc[i]))
    if np.any(np.isnan(g)):
        return 0
    else:
        return g

def gradient_descend_step1(obj_loc, d, sen_loc, step_count,
    step_size):
    sen_loc = sen_loc - step_size(step_count) * gradient1(obj_loc,
        d, sen_loc)
    return sen_loc

def gradient_descend1(obj_loc, d, sen_loc, total_step_count,
    step_size, err=0):
    positions = [np.array(sen_loc)]
    for k in range(total_step_count):
        new = gradient_descend_step1(obj_loc, d, positions[-1],
            k, step_size)
        if np.linalg.norm(positions[-1]-new)<err:
            break
        else:
            positions.append(new)
    return np.array(positions)

position4 = []
total_step_count = 10000  # number of GD steps to take
step_size = lambda i: 1/(1+i)#0.001  # step size at iteration i
```

Solution (cont.)

```python
err = 1e-10
for i in range(num_gd_replicates):
    p = []
    initial_position4 = np.random.randn(7,2)*100
    optimal_solution4 = gradient_descend1(train, train_distances,
        initial_position4, total_step_count, step_size, err)
    p += [optimal_solution4[-1]]
    position4 += [np.array(p)]
position4 = np.array(position4)
result4 = np.unique(np.round(position4,2),axis=0)[np.argmax(
    np.unique(np.round(position4,2),axis=0,return_counts=True)[1])]

def gradient2(obj_loc, d, sen_loc):
    g = 2*np.dot((((np.linalg.norm(obj_loc-sen_loc,axis=1)-d)/
      np.linalg.norm(obj_loc-sen_loc,axis=1))).T,(obj_loc-sen_loc))
    if np.any(np.isnan(g)):
        return 0
    else:
        return g/len(g)

def gradient_descend_step2(obj_loc, d, sen_loc, step_count,
    step_size):
    obj_loc = obj_loc - step_size(step_count) * gradient2(obj_loc,
        d, sen_loc)
    return obj_loc

def gradient_descend2(obj_loc, d, sen_loc, total_step_count,
    step_size, err=0):
    positions = [np.array(obj_loc)]
    for k in range(total_step_count):
        new = gradient_descend_step2(positions[-1], d, sen_loc,
            k, step_size)
        if np.max(np.linalg.norm(positions[-1]-new,axis=1))<err:
            break
        else:
            positions.append(new)
    return positions

total_step_count = 1000  # number of GD steps to take
step_size = lambda i: 1/(1+i)#0.001  # step size at iteration i
err = 1e-10

result5 = np.zeros_like(train)
```

Solution (cont.)

```
for j in range(len(train)):
    p = []
    for i in range(num_gd_replicates):
        initial_position5 = np.random.randn(1,2)*100
        optimal_solution5 = gradient_descend2(initial_position5,
            train_distances[j], result4[0], total_step_count,
              step_size, err)
        p += [optimal_solution5[-1]]
    position5 = np.array(p)
    result5[j] = np.unique(np.round(position5,2),axis=0)[np.argmax
      (np.unique(np.round(position5,2),axis=0,return_counts=True)
        [1])]

result51 = np.zeros_like(test1)
for j in range(len(test1)):
    p = []
    for i in range(num_gd_replicates):
        initial_position51 = np.random.randn(1,2)*100
        optimal_solution51 = gradient_descend2(initial_position51,
           test1_distances[j], result4[0], total_step_count,
            step_size, err)
        p += [optimal_solution51[-1]]
    position51 = np.array(p)
    result51[j] = np.unique(np.round(position51,2),axis=0)[
      np.argmax(np.unique(np.round(position51,2),axis=0,
      return_counts=True)[1])]

result52 = np.zeros_like(test2)
for j in range(len(test2)):
    p = []
    for i in range(num_gd_replicates):
        initial_position52 = np.random.randn(1,2)*100
        optimal_solution52 = gradient_descend2(initial_position52,
           test2_distances[j], result4[0], total_step_count,
            step_size, err)
        p += [optimal_solution52[-1]]
    position52 = np.array(p)
    result52[j] = np.unique(np.round(position52,2),axis=0)[
      np.argmax(np.unique(np.round(position52,2),axis=0,
      return_counts=True)[1])]

MSE_train = np.sum(np.linalg.norm(result5 - train,axis=1)**2)
  /len(train)
```

Solution (cont.)

```
MSE_test1 = np.sum(np.linalg.norm(result51 - test1,axis=1)**2)
  /len(test2)
MSE_test2 = np.sum(np.linalg.norm(result52 - test2,axis=1)**2)
  /len(test1)
```

Question 5

Have uploaded to Gradescope.

Question 6

Question What is the difference between various kinds of gradient descend method?
Solution
For cost function θ, data set D with m data, η is the learning rate

(1) Batch Gradient Descend (BGD)

$$\theta_j = \theta_j - \eta \frac{1}{m} \sum_{i=1}^{m} [h_\theta(x_i) - y_i] x_i$$

It cannot work well when m is very large.

(2) Mini-Batch Gradient Descend (MBGD)

Given batch size b, we only calculate the gradient for a small subset of D. We can split D into several subset or we can just choose each S randomly.

$$\theta_j = \theta_j - \eta \frac{1}{b} \sum_{\substack{S \subset D \\ |S| = b}} [h_\theta(x_i) - y_i] x_i$$

(3) Stochasitic Gradient Descend (SGD)

For $i = 1, 2, \cdots, m$,

$$\theta_j = \theta_j - \eta [h_\theta(x_i) - y_i] x_i$$

it may not return a most accuracy optimal solution but it is useful because it cost less time.

Method	BGD	MBGD	SGD
Accuracy	high	medium	low
Time consuming	low	medium	high