4B

```
import numpy as np
import scipy.spatial

################################################################################
#########  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):
        """
        Generate the locations of n points.

   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.

        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 = 100*np.random.randn(n, d)
        if not original_dist:
          obj_loc += 1000

        distance = scipy.spatial.distance.cdist(obj_loc,
                                                                 sensor_loc,
```

```
metric='euclidean')
          distance += np.random.randn(n, k)
          return obj_loc, distance
#####################################################################
# Starter code for Part (b)
#####################################################################
import math
def compute_gradient(distance, loc, sensor_loc):
    '''
    This function computes the gradient at a particular point
    '''
    total_x = 0
    total_y = 0
    x1 = loc[0]
    y1 = loc[1]

    for i in range(7):
        ai = sensor_loc[i][0]
        bi = sensor_loc[i][1]
        di = distance[i]
        total_x += 2 * (math.sqrt((ai - x1)**2 + (bi -y1)**2) - di) * (ai - x1)/math.sqrt((ai - x1)**2 +
(bi -y1)**2)
    for i in range(7):
        ai = sensor_loc[i][0]
        bi = sensor_loc[i][1]
        di = distance[i]
        total_y += 2 * (math.sqrt((ai - x1)**2 + (bi -y1)**2) - di) * (bi - y1)/math.sqrt((ai - x1)**2 +
(bi -y1)**2)
    gradient = np.array([total_x,total_y]) * -1
    return gradient



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


def compute_updates(distance, p, sensor_loc, 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 A, b,for determining the step size at step i
    """
    positions = [np.array(p)]
    for k in range(total_step_count):
        positions.append(compute_update(distance,  positions[-1], sensor_loc, k, step_size))
    return np.array(positions)

np.random.seed(0)
sensor_loc = generate_sensors()
obj_loc, distance = generate_data(sensor_loc)
single_distance = distance[0]


total_step_count = 1000
step_size = lambda i: .05
initial_position = np.array([0, 0])
positions = compute_updates(single_distance, initial_position, sensor_loc, total_step_count,
step_size)
#compute_gradient(single_distance, np.array([0, 0]), sensor_loc)
print(positions)
initial_position = np.random.rand(2)*100
positions = compute_updates(single_distance, initial_position, sensor_loc, total_step_count,
step_size)
#compute_gradient(single_distance, np.array([0, 0]), sensor_loc)
print(positions)
```

4C

```
import numpy as np
import scipy.spatial

########################################################################
#########  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_given_location(sensor_loc, obj_loc, k = 7, d = 2):
	"""
	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)
	return obj_loc, distance

##################################################################
# Starter code for Part (b)
##################################################################
```

```
import math
def compute_gradient(distance, loc, sensor_loc):
    '''
    This function computes the gradient at a particular point
    '''
    total_x = 0
    total_y = 0
    x1 = loc[0]
    y1 = loc[1]

    for i in range(7):
        ai = sensor_loc[i][0]
        bi = sensor_loc[i][1]
        di = distance[i]
        total_x += 2 * (math.sqrt((ai - x1)**2 + (bi -y1)**2) - di) * (ai - x1)/math.sqrt((ai - x1)**2 +
(bi -y1)**2)
    for i in range(7):
        ai = sensor_loc[i][0]
        bi = sensor_loc[i][1]
        di = distance[i]
        total_y += 2 * (math.sqrt((ai - x1)**2 + (bi -y1)**2) - di) * (bi - y1)/math.sqrt((ai - x1)**2 +
(bi -y1)**2)
    gradient = np.array([total_x,total_y]) * -1
    return gradient



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


def compute_updates(distance, p, sensor_loc, 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 A, b,for determining the step size at step i
    """
    positions = [np.array(p)]
    for k in range(total_step_count):
        positions.append(compute_update(distance,  positions[-1], sensor_loc, k, step_size))
    return np.array(positions)
```

```
# Sensor locations.

num_gd_replicates = 100

obj_locs = [[[i,i]] for i in np.arange(0,1000,100)]
np.random.seed(100)
total_step_count = 1000
step_size = lambda i: .05
est_pos = [list()] * 10

for k, obj_loc in enumerate(obj_locs):
    for j in range(num_gd_replicates):
        sensor_loc = generate_sensors()
        obj_loc, distance = generate_data_given_location(sensor_loc, obj_loc, k = 7, d = 2)
        single_distance = distance[0]
        initial_position = (obj_loc[0][0]+1)*np.random.randn(2)
        positions = compute_updates(single_distance, initial_position, sensor_loc,
total_step_count, step_size)
        print(positions[-1])
#for pos in est_pos:
#    print(pos)
```