

EECS 16A HW13

Bryan Ngo

2019-11-30

1 Mechanical: Projections

1.a

$$\text{proj}_{\mathbf{a}} \mathbf{b} = \frac{\langle \mathbf{a}, \mathbf{b} \rangle}{\|\mathbf{a}\|^2} \mathbf{a} = \frac{2}{1^2 + 0^2 + 1^2} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \quad (1)$$

$$\|\mathbf{e}\|^2 = \|\text{proj}_{\mathbf{a}} \mathbf{b} - \mathbf{b}\|^2 = \left\| \begin{bmatrix} 1-3 \\ 0-2 \\ 1+1 \end{bmatrix} \right\|^2 = \left\| \begin{bmatrix} -2 \\ -2 \\ 2 \end{bmatrix} \right\|^2 = 12 \quad (2)$$

1.b

Since the set of vectors are orthogonal to each other, then the projection upon the subspace is simply the sum of the projections of \mathbf{b} onto the two spanning vectors of the set $\text{colspace}(\mathbf{S})$.

$$\text{proj}_{\mathbf{S}_1} \mathbf{b} = \frac{1}{2} \left\langle \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 4 \\ -5 \end{bmatrix} \right\rangle \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} -2 \\ 0 \\ -2 \end{bmatrix} \quad (3)$$

$$\text{proj}_{\mathbf{S}_2} \mathbf{b} = \frac{1}{2} \left\langle \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 4 \\ -5 \end{bmatrix} \right\rangle \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \\ 0 \end{bmatrix} \quad (4)$$

$$\text{proj}_{\mathbf{S}} \mathbf{b} = \text{proj}_{\mathbf{S}_1} \mathbf{b} + \text{proj}_{\mathbf{S}_2} \mathbf{b} = \begin{bmatrix} -2 \\ 2 \\ -2 \end{bmatrix} \quad (5)$$

The error magnitude is

$$\|\mathbf{e}\|^2 = \|\text{proj}_{\mathbf{S}} \mathbf{b} - \mathbf{b}\|^2 = \left\| \begin{bmatrix} -2-1 \\ 2-4 \\ -2+5 \end{bmatrix} \right\|^2 = \left\| \begin{bmatrix} -3 \\ -2 \\ 3 \end{bmatrix} \right\|^2 = 22 \quad (6)$$

2 Mechanical: Least Squares

2.a

Linear regressions means we optimize the equation $mx + b = y$. We construct the matrix-vector multiplication with $b = 0$,

$$\underbrace{\begin{bmatrix} 2 \\ 4 \\ 6 \\ 8 \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} m \end{bmatrix}}_{\hat{\mathbf{x}}} = \underbrace{\begin{bmatrix} 2 \\ 6 \\ 7 \\ 8 \end{bmatrix}}_{\mathbf{b}} \quad (7)$$

Using least squares,

$$[\mathbf{A}]^\top \mathbf{A} = \|\mathbf{A}\|^2 = 2^2 + 4^2 + 6^2 + 8^2 = 120 \quad (8)$$

$$[\mathbf{A}]^\top \mathbf{b} = \langle \mathbf{A}, \mathbf{b} \rangle = 4 + 24 + 42 + 64 = 134 \quad (9)$$

$$\hat{\mathbf{x}} = ([\mathbf{A}]^\top \mathbf{A})^{-1} [\mathbf{A}]^\top \mathbf{b} = \frac{134}{120} \quad (10)$$

See Jupyter Notebook for plot. The error magnitude is

$$\|e\|^2 = \|\mathbf{A}\hat{\mathbf{x}} - \mathbf{b}\|^2 = \left\| \frac{134}{120} \begin{bmatrix} 2 \\ 4 \\ 6 \\ 8 \end{bmatrix} - \begin{bmatrix} 2 \\ 6 \\ 7 \\ 8 \end{bmatrix} \right\|^2 = \left\| \begin{bmatrix} 7/30 \\ -23/15 \\ -3/10 \\ 14/15 \end{bmatrix} \right\|^2 = \frac{101}{30} \quad (11)$$

2.b

We construct the matrix multiplication

$$\begin{bmatrix} 2 & 1 \\ 4 & 1 \\ 6 & 1 \\ 8 & 1 \end{bmatrix} \begin{bmatrix} m \\ b \end{bmatrix} = \begin{bmatrix} 2 \\ 6 \\ 7 \\ 8 \end{bmatrix} \quad (12)$$

Using least squares,

$$([\mathbf{A}]^\top \mathbf{A})^{-1} = \begin{bmatrix} 2 & 4 & 6 & 8 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 4 & 1 \\ 6 & 1 \\ 8 & 1 \end{bmatrix} = \begin{bmatrix} 120 & 20 \\ 20 & 4 \end{bmatrix}^{-1} = \begin{bmatrix} 1/20 & -1/4 \\ -1/4 & 3/2 \end{bmatrix} \quad (13)$$

$$[\mathbf{A}]^\top \mathbf{b} = \begin{bmatrix} 2 & 4 & 6 & 8 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 6 \\ 7 \\ 8 \end{bmatrix} = \begin{bmatrix} 134 \\ 23 \end{bmatrix} \quad (14)$$

$$\hat{\mathbf{x}} = ([\mathbf{A}]^\top \mathbf{A})^{-1} [\mathbf{A}]^\top \mathbf{b} = \begin{bmatrix} 1/20 & -1/4 \\ -1/4 & 3/2 \end{bmatrix} \begin{bmatrix} 134 \\ 23 \end{bmatrix} = \begin{bmatrix} 19/20 \\ 1 \end{bmatrix} \quad (15)$$

See Jupyter Notebook for plot. The error magnitude is

$$\|e\|^2 = \|\mathbf{A}\hat{\mathbf{x}} - \mathbf{b}\|^2 = \left\| \begin{bmatrix} 2 & 1 \\ 4 & 1 \\ 6 & 1 \\ 8 & 1 \end{bmatrix} \begin{bmatrix} 19/20 \\ 1 \end{bmatrix} - \begin{bmatrix} 2 \\ 6 \\ 7 \\ 8 \end{bmatrix} \right\|^2 \quad (16)$$

$$\mathbf{A}\hat{\mathbf{x}} - \mathbf{b} = \begin{bmatrix} 2 & 1 \\ 4 & 1 \\ 6 & 1 \\ 8 & 1 \end{bmatrix} \begin{bmatrix} 19/20 \\ 1 \end{bmatrix} - \begin{bmatrix} 2 \\ 6 \\ 7 \\ 8 \end{bmatrix} = \begin{bmatrix} 19/10 - 2 \\ 19/5 - 6 \\ 57/10 - 7 \\ 38/5 - 8 \end{bmatrix} = \begin{bmatrix} -1/10 \\ -11/5 \\ -13/10 \\ -2/5 \end{bmatrix} \quad (17)$$

$$\|\mathbf{A}\hat{\mathbf{x}} - \mathbf{b}\|^2 = \frac{67}{10} \quad (18)$$

This error magnitude is larger than the first, so the affine fit is *not* a better fit. In fact, it is worse by a factor of $\frac{\|e_{\text{aff}}\|^2}{\|e_{\text{lin}}\|^2} \approx 2$. Qualitatively, we can see this through the fact that the green line does not get as close to as many points as the red line.

2.c Prove the following theorem

Theorem 1. Let $\mathbf{A} = \mathbb{R}^{m,n}$. If $\hat{\mathbf{x}}$ solves the problem $\min_{\hat{\mathbf{x}}} \|\mathbf{A}\hat{\mathbf{x}} - \mathbf{b}\|^2$, then $\hat{\mathbf{x}}$ is orthogonal to $\text{colspace}(\mathbf{A})$.

Proof. By definition, $\hat{\mathbf{x}}$ being the solution to the least squares problem means that

$$\hat{\mathbf{x}} = ([\mathbf{A}]^\top \mathbf{A})^{-1} [\mathbf{A}]^\top \mathbf{b} \quad (19)$$

$$([\mathbf{A}]^\top \mathbf{A})\hat{\mathbf{x}} = [\mathbf{A}]^\top \mathbf{b} \quad (20)$$

$$([\mathbf{A}]^\top \mathbf{A})\hat{\mathbf{x}} - [\mathbf{A}]^\top \mathbf{b} = 0 \quad (21)$$

$$[\mathbf{A}]^\top (\mathbf{A}\hat{\mathbf{x}} - \mathbf{b}) = 0 \quad (22)$$

□

3 Trilateration with Noise!

3.a

$$(x - x_1)^2 + (y - y_1)^2 = d_1^2 \quad (23)$$

$$(x - x_2)^2 + (y - y_2)^2 = d_2^2 \quad (24)$$

$$(x - x_3)^2 + (y - y_3)^2 = d_3^2 \quad (25)$$

$$(x - x_4)^2 + (y - y_4)^2 = d_4^2 \quad (26)$$

3.b

$$(x - x_2)^2 - (x - x_1)^2 + (y - y_2)^2 - (y - y_1)^2 = d_2^2 - d_1^2 \quad (27)$$

$$(x - x_3)^2 - (x - x_1)^2 + (y - y_3)^2 - (y - y_1)^2 = d_3^2 - d_1^2 \quad (28)$$

$$(x - x_4)^2 - (x - x_1)^2 + (y - y_4)^2 - (y - y_1)^2 = d_4^2 - d_1^2 \quad (29)$$

Simplifying,

$$2x(x_1 - x_2) + x_2^2 - x_1^2 + 2y(y_1 - y_2) + y_2^2 - y_1^2 = d_2^2 - d_1^2 \quad (30)$$

$$2x(x_1 - x_3) + x_3^2 - x_1^2 + 2y(y_1 - y_2) + y_2^2 - y_1^2 = d_3^2 - d_1^2 \quad (31)$$

$$2x(x_1 - x_4) + x_4^2 - x_1^2 + 2y(y_1 - y_2) + y_2^2 - y_1^2 = d_4^2 - d_1^2 \quad (32)$$

3.c

$$\begin{bmatrix} 2(x_1 - x_2) & 2(y_1 - y_2) \\ 2(x_1 - x_3) & 2(y_1 - y_2) \\ 2(x_1 - x_4) & 2(y_1 - y_2) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} d_2^2 - d_1^2 + x_1^2 + y_1^2 - x_2^2 - y_2^2 \\ d_3^2 - d_1^2 + x_1^2 + y_1^2 - x_3^2 - y_2^2 \\ d_4^2 - d_1^2 + x_1^2 + y_1^2 - x_4^2 - y_2^2 \end{bmatrix} \quad (33)$$

3.d

The observer is at $(0, 0)$.

3.e

See Jupyter Notebook.

3.f

See Jupyter Notebook.

3.g

The plot of `imperfect_distances` makes sense given that the location is quite close to the intersections of the circles. However, I would intuitively not chosen the spot calculated in `one_bad_distances`. The code simply averages the distance between the bad distance and the other three good distances. Knowing there is only one erroneous measurement, I would've chosen the spot with the *most* intersections.

3.h

The cost algorithm returns the worst result when calculating `one_bad_distances`.

4 Labeling Patients Using Gene Expression Data

4.a

Our unknowns are $\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5$.

4.b

Using Jupyter Notebook, we find the vector

$$\boldsymbol{\alpha} \approx \begin{bmatrix} -0.16 \\ 0.09 \\ 0.48 \\ -0.58 \\ -0.35 \end{bmatrix} \quad (34)$$

4.c

The tests predict with 100% accuracy whether each mouse has diabetes or not.

5 Image Analysis

5.a

Suppose we plug in multiple values of (x_n, y_n) into the equation for a circle. We then get a system of equations linear in a, d, e that we can use to construct the matrix-vector multiplication

$$\begin{bmatrix} x_1^2 + y_1^2 & x_1 & y_1 \\ x_2^2 + y_2^2 & x_2 & y_2 \\ \vdots & \vdots & \vdots \\ x_n^2 + y_n^2 & x_n & y_n \end{bmatrix} \begin{bmatrix} a \\ d \\ e \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \quad (35)$$

where we simply plug in all the data points and use least squares to solve for our coefficients.

5.b

$$\begin{bmatrix} x_1^2 & x_1 y_1 & y_1^2 & x_1 & y_1 \\ x_2^2 & x_2 y_2 & y_2^2 & x_2 & y_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n^2 & x_n y_n & y_n^2 & x_n & y_n \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \quad (36)$$

5.c

We find $\frac{\|\mathbf{e}\|}{N} \approx 0.137$.

5.d

We find $\frac{\|e\|}{N} \approx 0.013$. The error is *much* lower, so I would prefer the ellipse fit over the circle fit.

6 Homework Process and Study Group

I did this homework by myself.

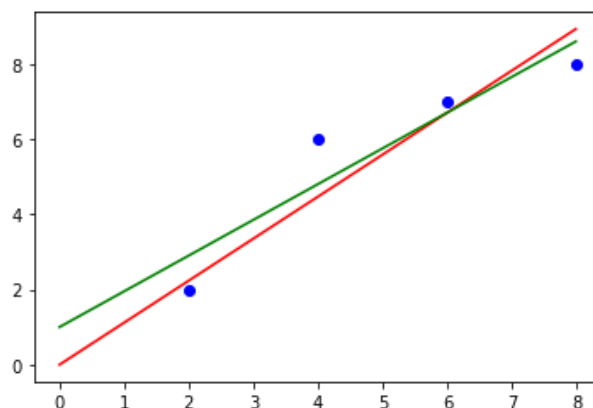
EECS16A: Homework 13

```
In [1]: from __future__ import division
%pylab inline
import numpy as np
import matplotlib.pyplot as plt
import scipy.io
import sys
```

Populating the interactive namespace from numpy and matplotlib

Mechanical Linear Least Squares (optional)

```
In [2]: # use for plotting
a = np.array([2,4,6,8])
b = np.array([2,6,7,8])
t = np.arange(0., 8., 0.01)
plt.plot(a, b, 'bo', t, 134/120 * t, 'r', t, 19/20 * t + 1, 'g')
plt.show()
```



Demonstration: Trilateration With Noise!

In lecture, we learned how the GPS receiver determines its location once it knows the distance of the various signaling beacons from itself. This method is called *trilateration*.

In this demonstration, we're going to further explore the connection between trilateration and least squares through a toy problem with four beacons and one GPS receiver.

We are given *three* possible sets of measurements for the distances of each of the beacons from the receiver:

1. First, the ideal set of measurements. $d_1 = d_2 = d_3 = d_4 = 5$.
2. Next, a set of imperfect measurements. $d_1 = 5, d_2 = 4.5, d_3 = 5, d_4 = 5.5$.
3. Finally, a set of mostly perfect measurements, but d_1 is a very bad measurement. We have $d_1 = 6.5$ and $d_2 = d_3 = d_4 = 5$.

First, we set up some notation for the positions of the beacons, as well as their respective distances from the receivers.

```
In [3]: from utils import *

ideal_distances = [5, 5, 5, 5]
imperfect_distances = [5.5, 4.5, 5, 5]
one_bad_distances = [6.5, 5, 5, 5]

#these are the coordinates of the beacons
positions = np.array([
    [0, -5],
    [-5 / 2, 5 * np.sqrt(3) / 2],
    [0, 5],
    [5 / 2, 5 * np.sqrt(3) / 2],
])

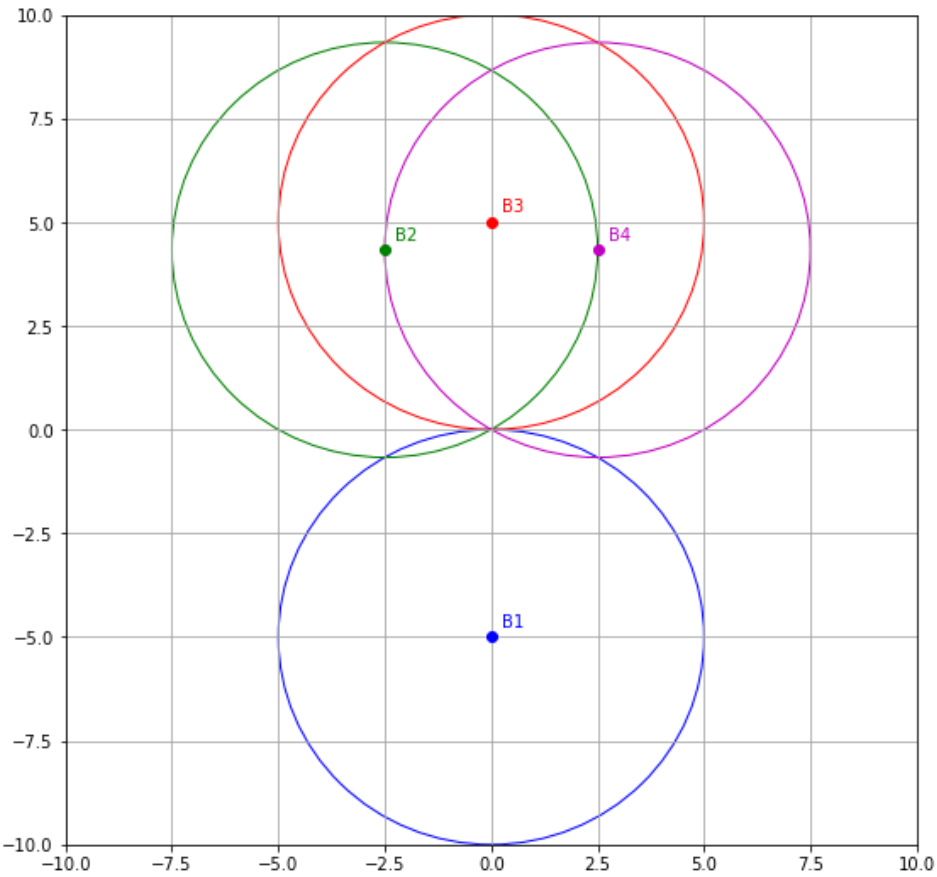
xpositions = positions[:, 0]
ypositions = positions[:, 1]

# setup to make the helper functions work
register(positions=positions)
register(xpositions=xpositions)
register(ypositions=ypositions)
```

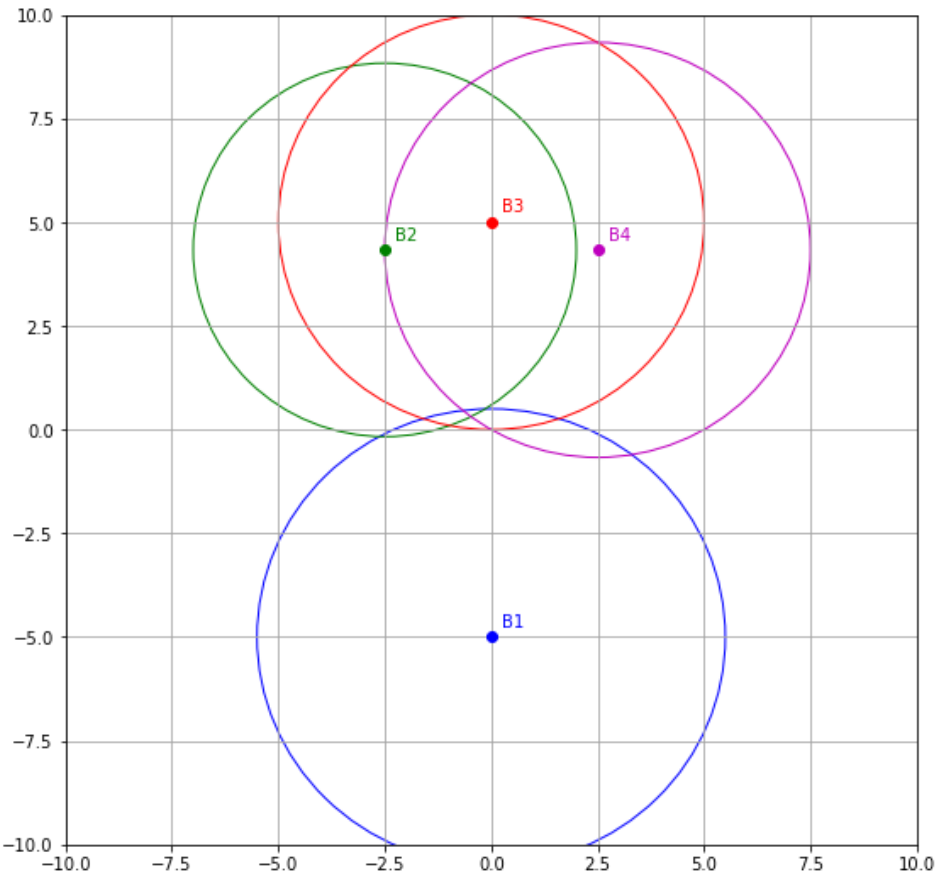
Part (d)

Now, for each of the three above cases, let's plot a circle for each beacon whose radius corresponds to the reported distance of the receiver. The intersection of the circles will tell us, intuitively, where the receiver is located! (Note that the circles do not necessarily intersect at one point for all three of the cases. Think about what this means).

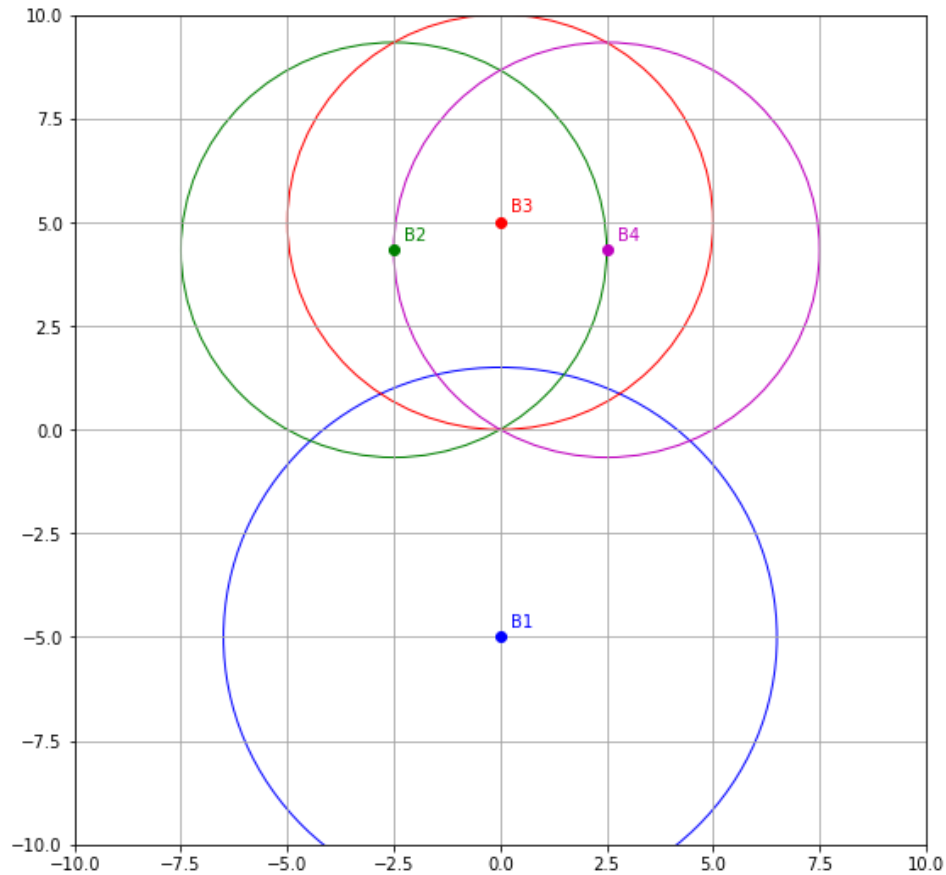

```
In [4]: plot(ideal_distances)
```



```
In [5]: plot(imperfect_distances)
```



```
In [6]: plot(one_bad_distances)
```



Part (e)

Now, let's solve for the location of the receiver, $\begin{bmatrix} x \\ y \end{bmatrix}$, using least squares.

Recall that we made the system of equations linear by subtracting the equation for the first beacon from each of the equations for the other beacons.

This will result in the system of equations:

$$A \begin{bmatrix} x \\ y \end{bmatrix} = \vec{b}$$

You will define A and \vec{b} in the code blocks below.

```
In [7]: A = np.zeros((3,2))

A[0][0] = 2*(xpositions[0] - xpositions[1])
A[0][1] = 2*(ypositions[0] - ypositions[1])
A[1][0] = 2*(xpositions[0] - xpositions[2])
A[1][1] = 2*(ypositions[0] - ypositions[2])
A[2][0] = 2*(xpositions[0] - xpositions[3])
A[2][1] = 2*(ypositions[0] - ypositions[3])
print(A)

[[ 5.         -18.66025404]
 [ 0.         -20.         ]
 [-5.         -18.66025404]]
```

Part (f)

Fill in the entries of `b` in the below function to correspond to the entries of \vec{b} from the problem.

```
In [8]: def make_b(distances):
        """
        Since `b` depends on `distances`, we implement it using a function, so we c
        an generate
        different `b` vectors depending of which set of distances we are interested
        in,
        (i.e. ideal_distances, imperfect_distances, or one_bad_distances)

        Examples of how to call the function:
        make_values(ideal_distances) OR
        make_values(imperfect_distances) OR
        make_values(one_bad_distances)
        """

        b = np.zeros(3)

        b[0] = distances[1]**2 - distances[0]**2 + xpositions[0]**2 + ypositions[0]
        **2 - xpositions[1]**2 - ypositions[1]**2
        b[1] = distances[2]**2 - distances[0]**2 + xpositions[0]**2 + ypositions[0]
        **2 - xpositions[2]**2 - ypositions[2]**2
        b[2] = distances[3]**2 - distances[0]**2 + xpositions[0]**2 + ypositions[0]
        **2 - xpositions[3]**2 - ypositions[3]**2

        return b

make_b(ideal_distances)
```

```
Out[8]: array([3.55271368e-15, 0.00000000e+00, 3.55271368e-15])
```

Part (g)

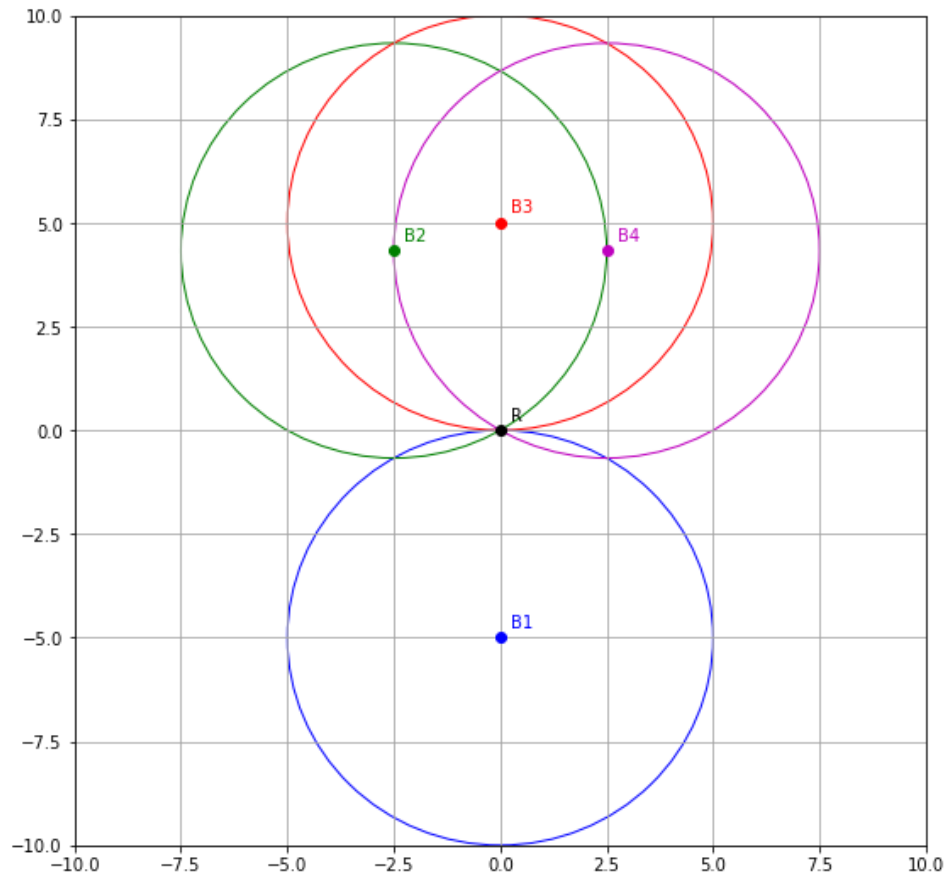
Now, calculate the linear least squares estimate for the `ideal_distances` data and plot the results. We have given you code for the implementation for this part.

```
In [9]: def estimate_position(distances):
        U1 = np.dot(A.T, A)
        U2 = np.dot(A.T, make_b(distances))
        least_squares_sol = np.dot(np.linalg.inv(U1), U2)
        return least_squares_sol

        estimate_position(ideal_distances)
```

```
Out[9]: array([ 8.59260618e-33, -1.20930181e-16])
```

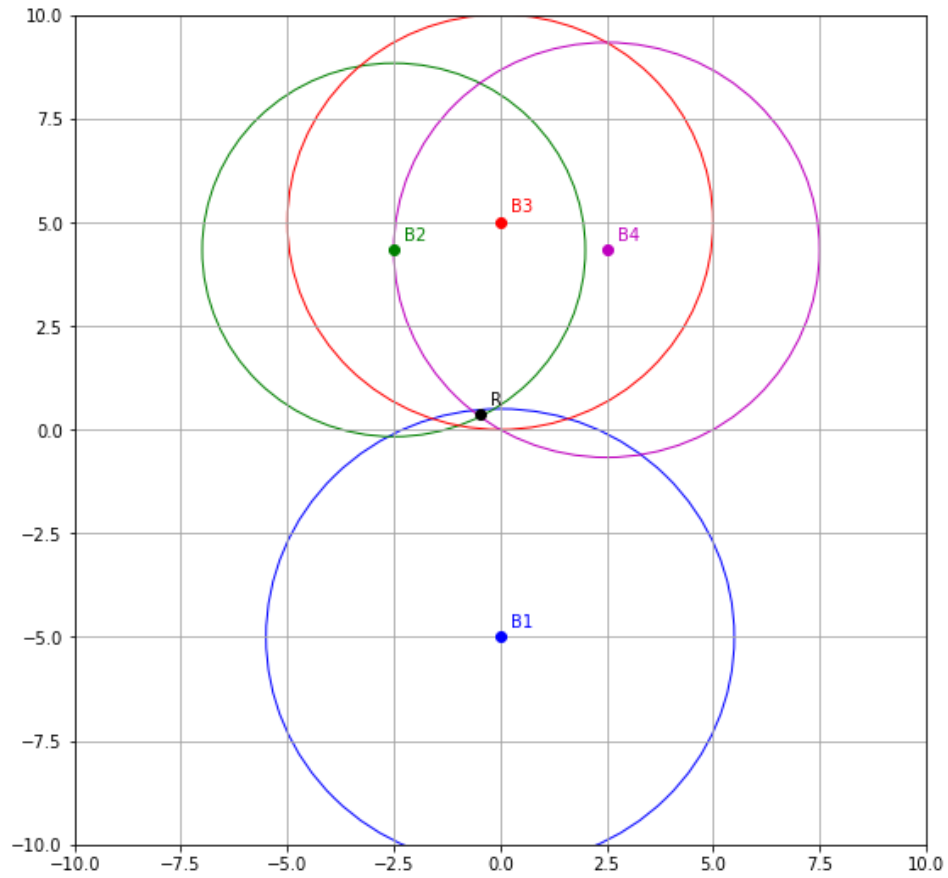
```
In [10]: plot(ideal_distances)
         plot_point(estimate_position(ideal_distances))
```



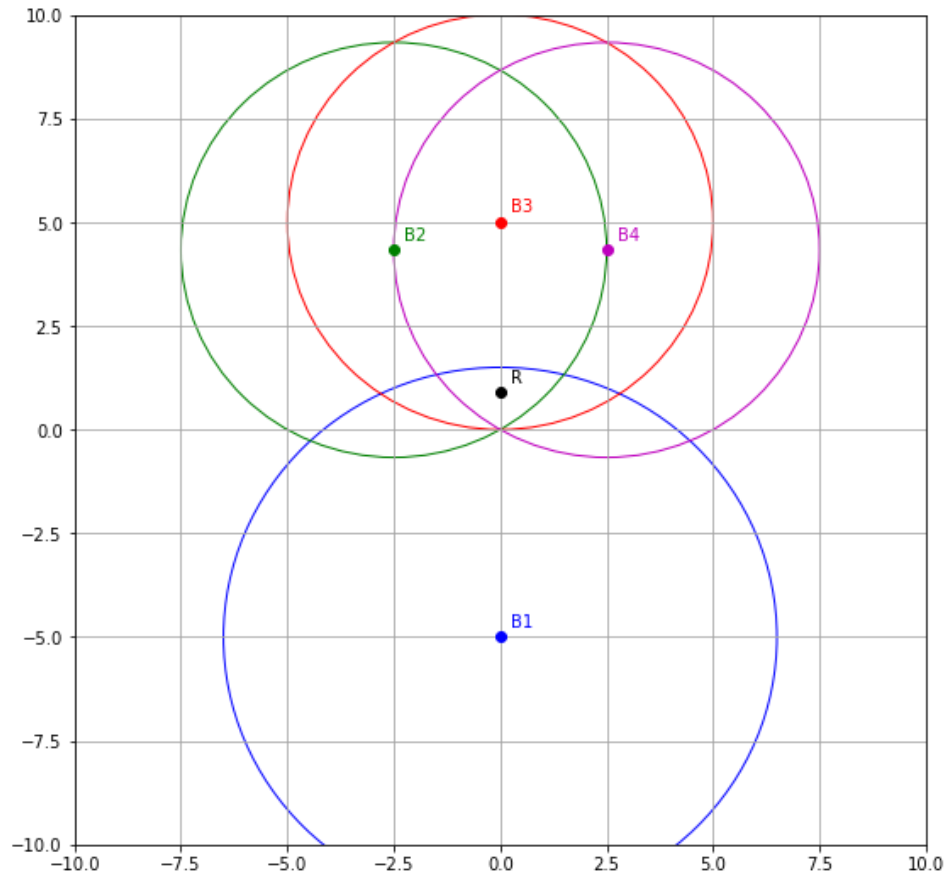
What about for the `imperfect_distances` and `one_bad_distances` ? Copy and modify the above code to compute and plot the least-squares trilateration solutions for those two cases, and comment on the quality of the solution in each case.

In particular, for `one_bad_distances` , is the solution the best possible? In other words, if you were trying to identify your position from the graph by hand, would you have chosen the same point that our trilateration solution did?

```
In [11]: ### imperfect_distances case  
  
#YOUR CODE HERE  
estimate_position(imperfect_distances)  
  
plot(imperfect_distances)  
plot_point(estimate_position(imperfect_distances))
```



```
In [12]: ### one_bad_distances case  
  
#YOUR CODE HERE  
estimate_position(one_bad_distances)  
  
plot(one_bad_distances)  
plot_point(estimate_position(one_bad_distances))
```



Part (h)

You should see that linearizing and solving least squares did not always do well in the above cases. Now, let's try something else. For any candidate location of the receiver, we can define the distance from i_{th} beacon as \hat{d}_i and define $\vec{\hat{d}}$ as:

$$\vec{\hat{d}} = \begin{bmatrix} \hat{d}_1 \\ \hat{d}_2 \\ \hat{d}_3 \\ \hat{d}_4 \end{bmatrix}$$

So we might want to minimize the following:

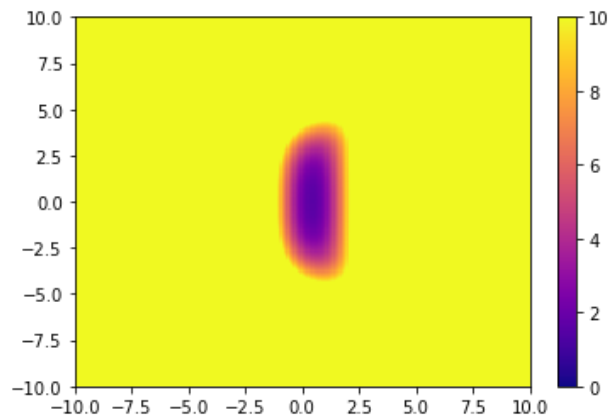
$$\sum_{i=1}^4 (d_i - \hat{d}_i)^2$$

Let's make a plot that will help us visualize this. (x, y) in the plot correspond to candidate positions of the receivers and the color corresponds to the cost.

Let's first plot the costs when the `one_bad_distances` data is used. You should see that this cost is minimized at $(0, 0)$.

```
In [13]: def cost_function(point, actual_distances):
    cost = 0
    for pos, actual_distance in zip(positions, actual_distances):
        cost += (distance_between_points(point, pos) - actual_distance) ** 2
    return cost
```

```
plot_cost(one_bad_distances, cost_function)
```



Let's verify numerically that the above approach does indeed do better than the least-squares trilateration approach in the `one_bad_distances` case. Use `cost_function` to compare the cost of $(0, 0)$ with the cost of your estimated position obtained from the least-squares solution in all three cases. When does least squares do worst, compared to the new approach?


```
In [14]: #YOUR CODE HERE
print(cost_function((0, 0), ideal_distances))
print(cost_function(estimate_position(imperfect_distances), ideal_distances))
print(cost_function(estimate_position(one_bad_distances), ideal_distances))

0.0
0.5430958151415356
2.772649125978342
```

Labeling Patients

```
In [15]: import numpy as np
```

```
In [16]: # Part B
A = np.load('gene_data_train.npy')
b = np.load('diabetes_train.npy')
x = np.linalg.lstsq(A, b, None)[0]
print(x)
```

```
[[ -0.15646169]
 [  0.09239418]
 [  0.48053974]
 [ -0.5847018 ]
 [ -0.35350734]]
```

```
In [17]: # Part C
# You may find np.sign useful to generate your +/- 1 prediction vector
A_test = np.load('gene_data_test.npy')
b_test = np.load('diabetes_test.npy')

prediction = np.sign(np.dot(A_test, x))
print(prediction)
print(b_test)
```

```
[[ 1.]
 [-1.]
 [-1.]
 [ 1.]]
[[ 1]
 [-1]
 [-1]
 [ 1]]
```

Image Analysis

```
In [18]: def plot_circle(a, d, e):
        """
        You can use this function to plot circles with parameters a,d,e.
        The parameters are described in the homework pdf.

        """
        is_circle = d**2 + e**2 - 4*a > 0
        assert is_circle, "Not a circle"

        XLIM_LO = -1
        XLIM_HI = 3
        YLIM_LO = -2
        YLIM_HI = 2
        X_COUNT = 400
        Y_COUNT = 400

        x = np.linspace(XLIM_LO, XLIM_HI, X_COUNT)
        y = np.linspace(YLIM_LO, YLIM_HI, Y_COUNT)
        x, y = np.meshgrid(x, y)
        f = lambda x,y: a*(x**2 + y**2) + d*x + e*y

        c1 = plt.contour(x, y, f(x,y), [1], colors='r')
        plt.axis('scaled')
        plt.xlabel('x')
        plt.ylabel('y')
        plt.title("Best Fit Circle")
```

```
In [19]: def plot_ellipse(a, b, c, d, e):
        """
        You can use this function to plot ellipses with parameters a-e.
        The parameters are described in the homework pdf.

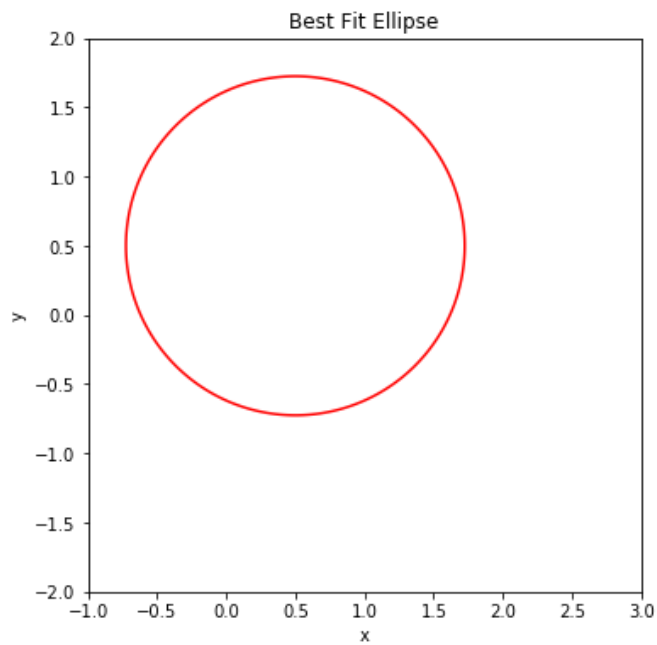
        """
        is_ellipse = b**2 - 4*a*c < 0
        assert is_ellipse, "Not an ellipse"

        XLIM_LO = -1
        XLIM_HI = 3
        YLIM_LO = -2
        YLIM_HI = 2
        X_COUNT = 400
        Y_COUNT = 400

        x = np.linspace(XLIM_LO, XLIM_HI, X_COUNT)
        y = np.linspace(YLIM_LO, YLIM_HI, Y_COUNT)
        x, y = np.meshgrid(x, y)
        f = lambda x,y: a*x**2 + b*x*y + c*y**2 + d*x + e*y

        c1 = plt.contour(x, y, f(x,y), [1], colors='r')
        plt.axis('scaled')
        plt.xlabel('x')
        plt.ylabel('y')
        plt.title("Best Fit Ellipse")
```

```
In [20]: # Here is an example of plot_ellipse.  
# This plots  $(x-1)**2 + (y-1)**2 = 1$ ,  
# which is a circle centered at (1,1).  
  
plt.figure(figsize=(6,6))  
plot_ellipse(1, 0, 1, -1, -1)
```



You may find [plt.scatter](http://matplotlib.org/api/pyplot_api.html) (http://matplotlib.org/api/pyplot_api.html) useful for plotting the points.

In [21]: *# PART C*

```

xy = np.array([[0.3, -0.69],
               [0.5, 0.87],
               [0.9, -0.86],
               [1, 0.88],
               [1.2, -0.82],
               [1.5, .64],
               [1.8, 0]])

x = xy[:,0]
y = xy[:,1]

# plot the data points
plt.scatter(x,y)

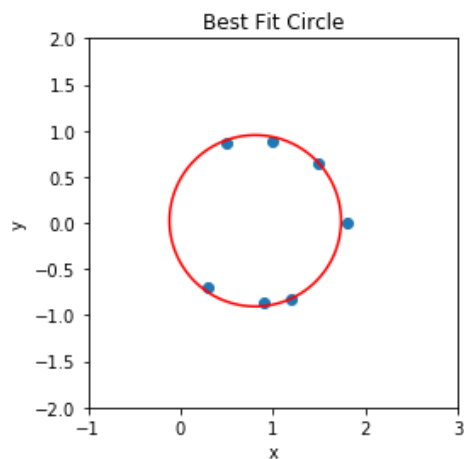
A = np.array([x**2+y**2,x,y]).T # Hint: this generates the A matrix
b = np.array([1 for i in range(7)])

circle_params = np.linalg.lstsq(A, b, None)[0]
circle_error = np.dot(A, circle_params) - b
error_mag = np.linalg.norm(circle_error)
print(error_mag / 7)
print(circle_params)
plot_circle(circle_params[0],circle_params[1],circle_params[2])

```

0.1374905622431481

[4.87314137 -7.89293482 -0.22651484]



```
In [22]: # PART D

# plot the data points
plt.scatter(x,y)

A = np.array([x**2, x*y, y**2, x, y]).T
b = np.array([1 for i in range(7)])

ellipse_params = np.linalg.lstsq(A, b, None)[0]
ellipse_error = np.dot(A, ellipse_params) - b
error_mag = np.linalg.norm(ellipse_error)
print(error_mag / 7)
print(ellipse_params)
plot_ellipse(ellipse_params[0],ellipse_params[1],ellipse_params[2],ellipse_params[3],ellipse_params[4])
```

0.012853829087236146

[4.10382951 0.48711384 4.93938449 -6.85032284 -0.62259259]

