1. Projections

a) $\text{proj}_{\vec{a}}(\vec{b}) = \frac{\langle \vec{b}, \vec{a} \rangle}{\|\vec{a}\|^2} \vec{a} = \frac{3+0-1}{(\sqrt{1+0+1})^2} \vec{a} = \frac{2}{2} \vec{a} = \vec{a}$

$$\text{proj}_{\vec{a}}(\vec{b}) = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

$$\|\vec{e}\|^2 = \|\text{proj}_{\vec{a}}(\vec{b}) - \vec{b}\|^2 = \left\| \begin{bmatrix} -2 \\ -2 \\ 2 \end{bmatrix} \right\|^2 = \sqrt{4+4+4}^{\,2} = \boxed{12}$$

b) $\text{proj}_{\left[\begin{smallmatrix}1\\0\\1\end{smallmatrix}\right]}(\vec{b}) + \text{proj}_{\left[\begin{smallmatrix}0\\1\\0\end{smallmatrix}\right]}(\vec{b}) = \frac{-4}{2} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} + \frac{4}{1} \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} =$

$$= \begin{bmatrix} -2 \\ 4 \\ -2 \end{bmatrix}$$

$$\|\vec{e}\|^2 = \left\| \begin{bmatrix} -3 \\ 0 \\ 3 \end{bmatrix} \right\|^2 = \boxed{18}$$

2. Least Squares

a) $\vec{a}x = \vec{b}$

$$x = \frac{\langle \vec{a}, \vec{b} \rangle}{\|\vec{a}\|^2} = \frac{4+24+42+64}{4+16+36+64} = 1.11667 \approx \boxed{1.117}$$

$$\|\vec{e}\|^2 = \|\vec{a}x - \vec{b}\|^2 = \left\| \begin{bmatrix} 2 \\ 4 \\ 6 \\ 8 \end{bmatrix} 1.117 - \begin{bmatrix} 2 \\ 6 \\ 7 \\ 8 \end{bmatrix} \right\|^2 = \left\| \begin{bmatrix} 0.233 \\ -1.533 \\ -0.3 \\ 0.933 \end{bmatrix} \right\|^2 = \boxed{3.367}$$

b) $\vec{a}x_1 + x_2 = \vec{b}$

$$\begin{bmatrix} 2 & 1 \\ 4 & 1 \\ 6 & 1 \\ 8 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 6 \\ 7 \\ 8 \end{bmatrix}$$

$$A \cdot \vec{x} = \vec{b}$$

$$\vec{x} = (A^T A)^{-1} A^T \vec{b} = \left(\begin{bmatrix} 2 & 4 & 6 & 8 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 4 & 1 \\ 6 & 1 \\ 8 & 1 \end{bmatrix} \right)^{-1} \begin{bmatrix} 2 & 4 & 6 & 8 \\ 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 2 \\ 6 \\ 7 \\ 8 \end{bmatrix} =$$

$$\vec{x} = \begin{bmatrix} 120 & 20 \\ 20 & 4 \end{bmatrix}^{-1} \begin{bmatrix} 134 \\ 23 \end{bmatrix} = \begin{bmatrix} 0.05 & -0.25 \\ -0.25 & 1.5 \end{bmatrix} \begin{bmatrix} 134 \\ 23 \end{bmatrix} = \begin{bmatrix} 0.95 \\ 1 \end{bmatrix}$$

$$\underline{\vec{a}(0.95) + 1 = \vec{b}}$$

$$\|\vec{e}\|^2 = \|\vec{a}x_1 + \vec{x}_2 - \vec{b}\|^2 = \left\| \begin{bmatrix} 0.9 \\ -1.2 \\ 0.3 \\ 0.6 \end{bmatrix} \right\|^2 = \boxed{2.7}$$

Yes, it is better for the data. This is because the error in part (b) is lower than error in part (a), which means model in part (b) fits better

3. Trilateration with Noise

a)

$$\|(x,y)-(x_1,y_1)\|^2 = d_1^2$$
$$\|(x,y)-(x_2,y_2)\|^2 = d_2^2$$
$$\|(x,y)-(x_3,y_3)\|^2 = d_3^2$$
$$\|(x,y)-(x_4,y_4)\|^2 = d_4^2$$

b) Let $\vec{p} = (x,y)$

$\vec{p}_1 = (x_1,y_1)$, $\vec{p}_2 = (x_2,y_2)$, $\vec{p}_3 = (x_3,y_3)$, $\vec{p}_4 = (x_4,y_4)$

$$\begin{aligned}
\langle \vec{p}-\vec{p}_1, \vec{p}-\vec{p}_1 \rangle &= d_1^2 \\
\langle \vec{p}-\vec{p}_2, \vec{p}-\vec{p}_2 \rangle &= d_2^2 \\
\langle \vec{p}-\vec{p}_3, \vec{p}-\vec{p}_3 \rangle &= d_3^2 \\
\langle \vec{p}-\vec{p}_4, \vec{p}-\vec{p}_4 \rangle &= d_4^2
\end{aligned}
\implies
\begin{aligned}
\|\vec{p}\|^2 - 2\langle \vec{p},\vec{p}_1\rangle + \|\vec{p}_1\|^2 &= d_1^2 \\
\|\vec{p}\|^2 - 2\langle \vec{p}_1,\vec{p}_2\rangle + \|\vec{p}_2\|^2 &= d_2^2 \\
\|\vec{p}\|^2 - 2\langle \vec{p},\vec{p}_3\rangle + \|\vec{p}_3\|^2 &= d_3^2 \\
\|\vec{p}\|^2 - 2\langle \vec{p},\vec{p}_4\rangle + \|\vec{p}_4\|^2 &= d_4^2
\end{aligned}$$

$$2\langle \vec{p},\vec{p}_1\rangle - 2\langle \vec{p},\vec{p}_2\rangle + \|\vec{p}_2\|^2 - \|\vec{p}_1\|^2 = d_2^2 - d_1^2$$
$$2\langle \vec{p},\vec{p}_1\rangle - 2\langle \vec{p},\vec{p}_3\rangle + \|\vec{p}_3\|^2 - \|\vec{p}_1\|^2 = d_3^2 - d_1^2$$
$$2\langle \vec{p},\vec{p}_1\rangle - 2\langle \vec{p},\vec{p}_4\rangle + \|\vec{p}_4\|^2 - \|\vec{p}_1\|^2 = d_4^2 - d_1^2$$

c)

$$2\vec{p}\,(\vec{p}_1-\vec{p}_2)^T = d_2^2 - d_1^2 - \|\vec{p}_2\|^2 + \|\vec{p}_1\|^2$$
$$2\vec{p}\,(\vec{p}_1-\vec{p}_3)^T = d_3^2 - d_1^2 - \|\vec{p}_3\|^2 + \|\vec{p}_1\|^2$$
$$2\vec{p}\,(\vec{p}_1-\vec{p}_4)^T = d_4^2 - d_1^2 - \|\vec{p}_4\|^2 + \|\vec{p}_1\|^2$$

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

$$\begin{bmatrix} 2((x_1,y_1)-(x_2,y_2))^T \\ 2((x_1,y_1)-(x_3,y_3))^T \\ 2((x_1,y_1)-(x_4,y_4))^T \end{bmatrix}\begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} d_2^2 - d_1^2 - \|(x_2,y_2)\|^2 + \|(x_1,y_1)\|^2 \\ d_3^2 - d_1^2 - \|(x_3,y_3)\|^2 + \|(x_1,y_1)\|^2 \\ d_4^2 - d_1^2 - \|(x_4,y_4)\|^2 + \|(x_1,y_1)\|^2 \end{bmatrix}$$

$$\begin{bmatrix} 2x_1-2x_2 & 2y_1-2y_2 \\ 2x_1-2x_3 & 2y_1-2y_3 \\ 2x_1-2x_4 & 2y_1-2y_4 \end{bmatrix}\begin{bmatrix} x \\ Y \end{bmatrix} = \begin{bmatrix} d_2^2 - d_1^2 - \|(x_2,y_2)\|^2 + \|(x_1,y_1)\|^2 \\ d_3^2 - d_1^2 - \|(x_3,y_3)\|^2 + \|(x_1,y_1)\|^2 \\ d_4^2 - d_1^2 - \|(x_4,y_4)\|^2 + \|(x_1,y_1)\|^2 \end{bmatrix}$$

3. d) The observer is at coordinate (0,0).

4. Labeling Patients using Gene Expression Data

a) Unknowns: α_1, α_2, α_3, α_4, α_5

5. Image Analysis

a) $a(x^2+y^2)+dx+ey=1$

$A\vec{x}=\vec{b}$

$$\begin{cases} a(0.3^2+0.69^2)+0.3d-0.69e=1 \\ a(0.5^2+0.87^2)+0.5d+0.87e=1 \\ a(0.9^2+0.86^2)+0.9d-0.86e=1 \\ a(1^2+0.88^2)+d+0.88e=1 \\ a(1.2^2+0.82^2)+1.2d-0.82e=1 \\ a(1.5^2+0.64^2)+1.5d+0.64e=1 \\ a(1.8^2)+1.8d=1 \end{cases}$$

$$\underbrace{\begin{bmatrix} 0.566 & 0.3 & -0.69 \\ 1.007 & 0.5 & 0.87 \\ 1.55 & 0.9 & -0.86 \\ 1.774 & 1 & 0.88 \\ 2.112 & 1.2 & -0.82 \\ 2.66 & 1.5 & 0.64 \\ 3.24 & 1.8 & 0 \end{bmatrix}}_{A} \cdot \underbrace{\begin{bmatrix} a \\ d \\ e \end{bmatrix}}_{\vec{x}} = \underbrace{\begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}}_{\vec{b}}$$

$\vec{x}=(A^TA)^{-1}A^T\vec{b}$

b) Similar to part (a), we will find A matrix.

$A\vec{x} = \vec{b}$

$$\begin{bmatrix} 0.3^2 & (0.3)(-0.69) & 0.69^2 & 0.3 & -0.69 \\ 0.5^2 & (0.5)(0.87) & 0.87^2 & 0.5 & 0.87 \\ 0.9^2 & (0.9)(-0.86) & 0.86^2 & 0.9 & -0.86 \\ 1 & 0.88 & 0.88^2 & 1 & 0.88 \\ 1.2^2 & (1.2)(-0.82) & 0.82^2 & 1.2 & -0.82 \\ 1.5^2 & (1.5)(0.64) & 0.64^2 & 1.5 & 0.64 \\ 1.8^2 & 0 & 0 & 1.8 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \\ e \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

$\vec{X} = (A^T A)^{-1} A^T \vec{b}$

d) Error in part (d) is lower than error in part (c). Because of that, the ellipse technique is better than the circle method.

6. Homework Process

I worked on this homework alone. I read all the notes and watched the lectures, so I was able to do them.

EECS16A: Homework 13

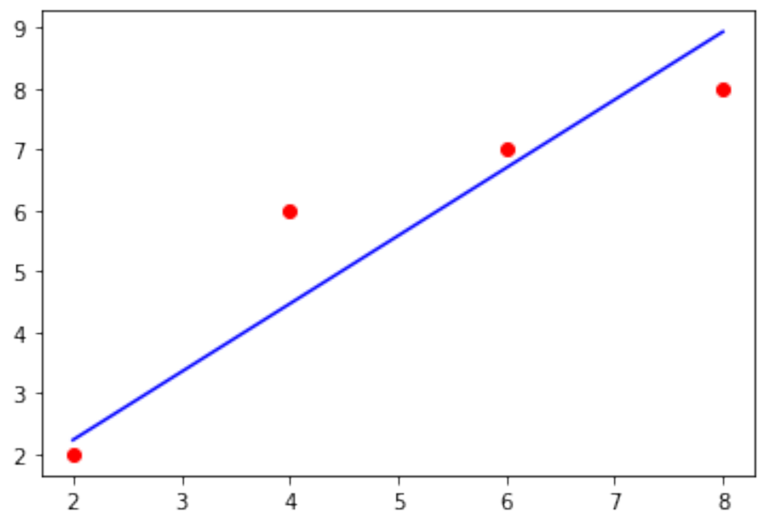
In [1]:
```
from __future__ import division
%pylab inline
import numpy as np
import matplotlib.pyplot as plt
import scipy.io
import sys
```

```
Populating the interactive namespace from numpy and matplotlib
```

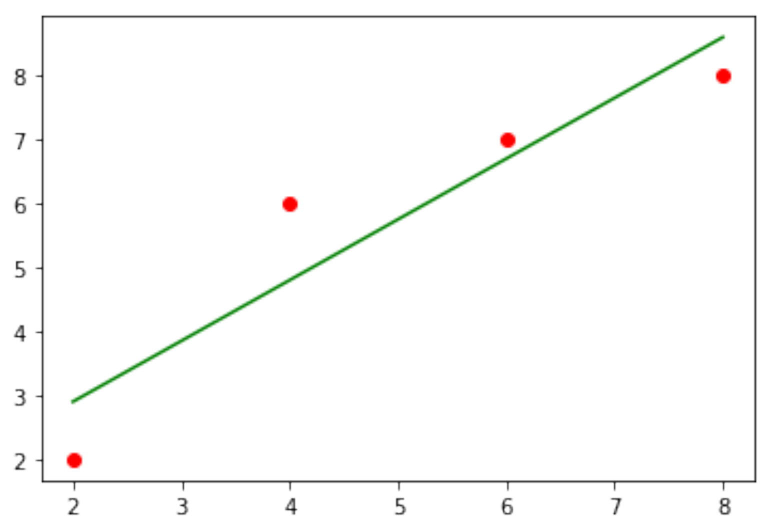
Mechanical Linear Least Squares (optional)

In [15]:
```
# use for plotting
print("part A")
a = np.array([2,4,6,8])
b = np.array([2,6,7,8])
for i in range(len(a)):
    plt.plot(a[i], b[i], color="red", marker = 'o')
plot(a, a*1.116666667, 'r', color="blue")
plt.show()

print("part B")
for i in range(len(a)):
    plt.plot(a[i], b[i], color="red", marker = 'o')
plot(a, a*0.95+1, 'r', color="green")
plt.show()
```

part A



part B



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 $d_2 = d_3 = d_4 = 5$.

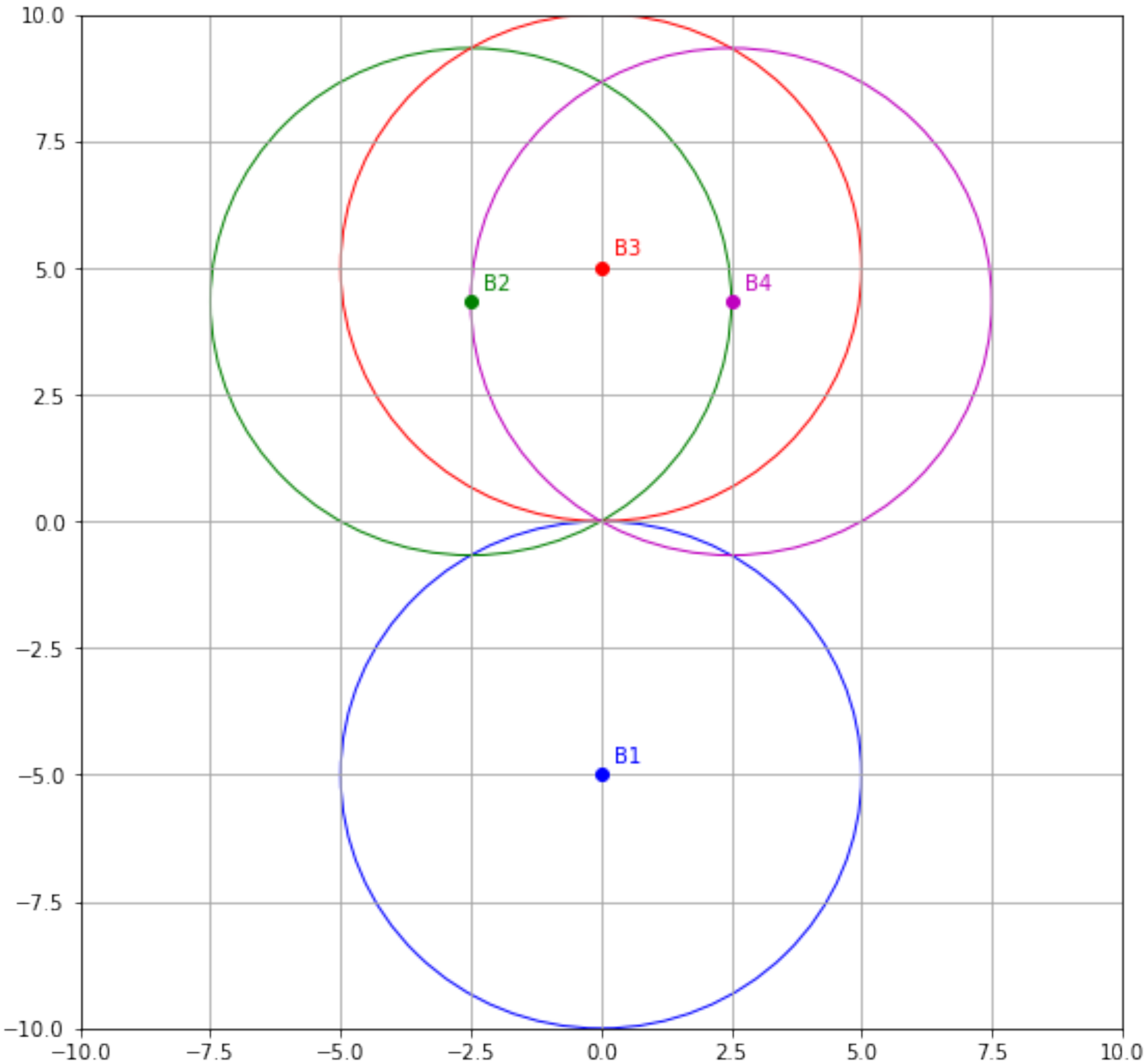
First, we set up some notation for the positions of the beacons, as well as their respective distances from the receivers.

In [16]:
```
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)
```

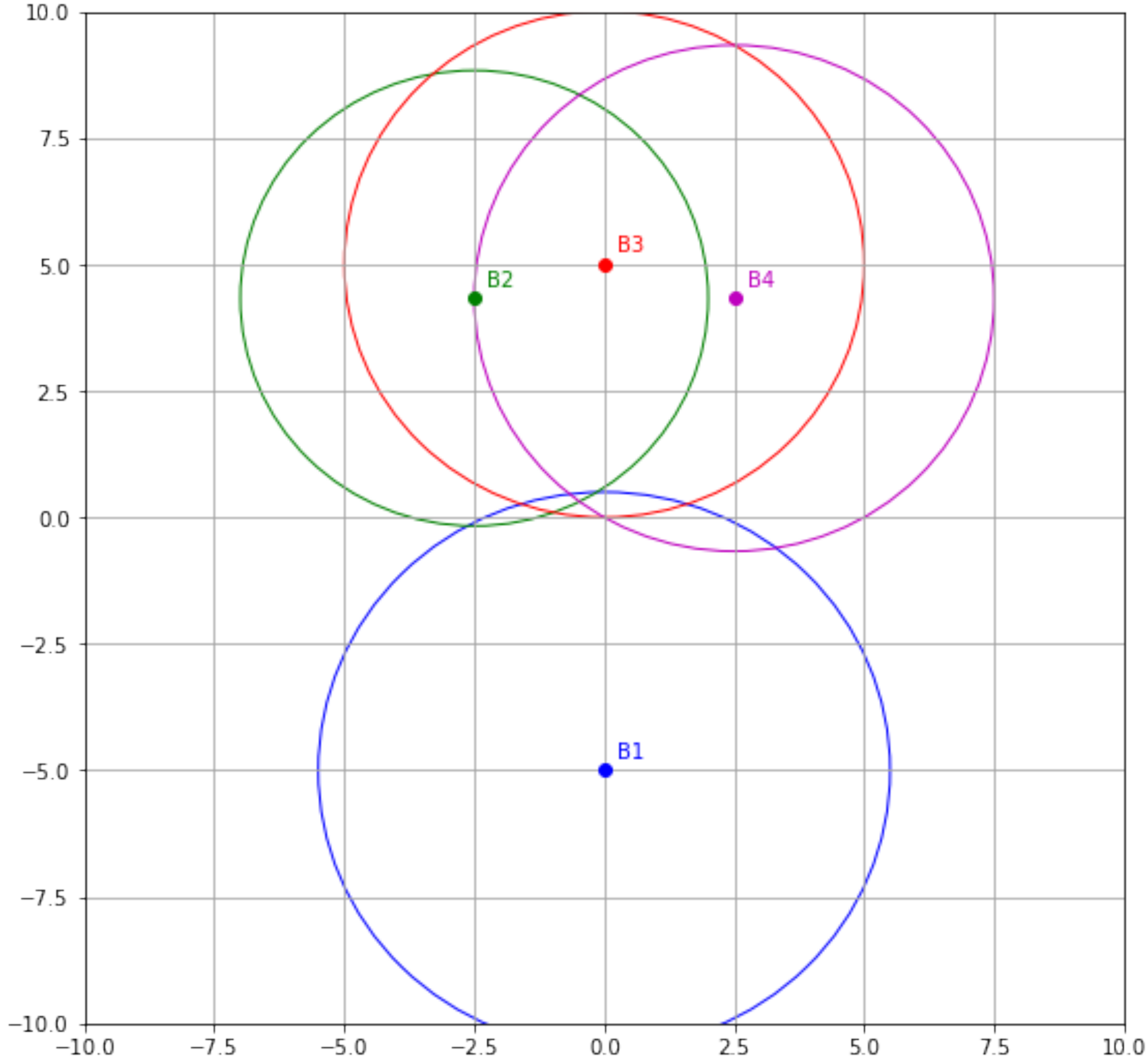
Part (d)

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).

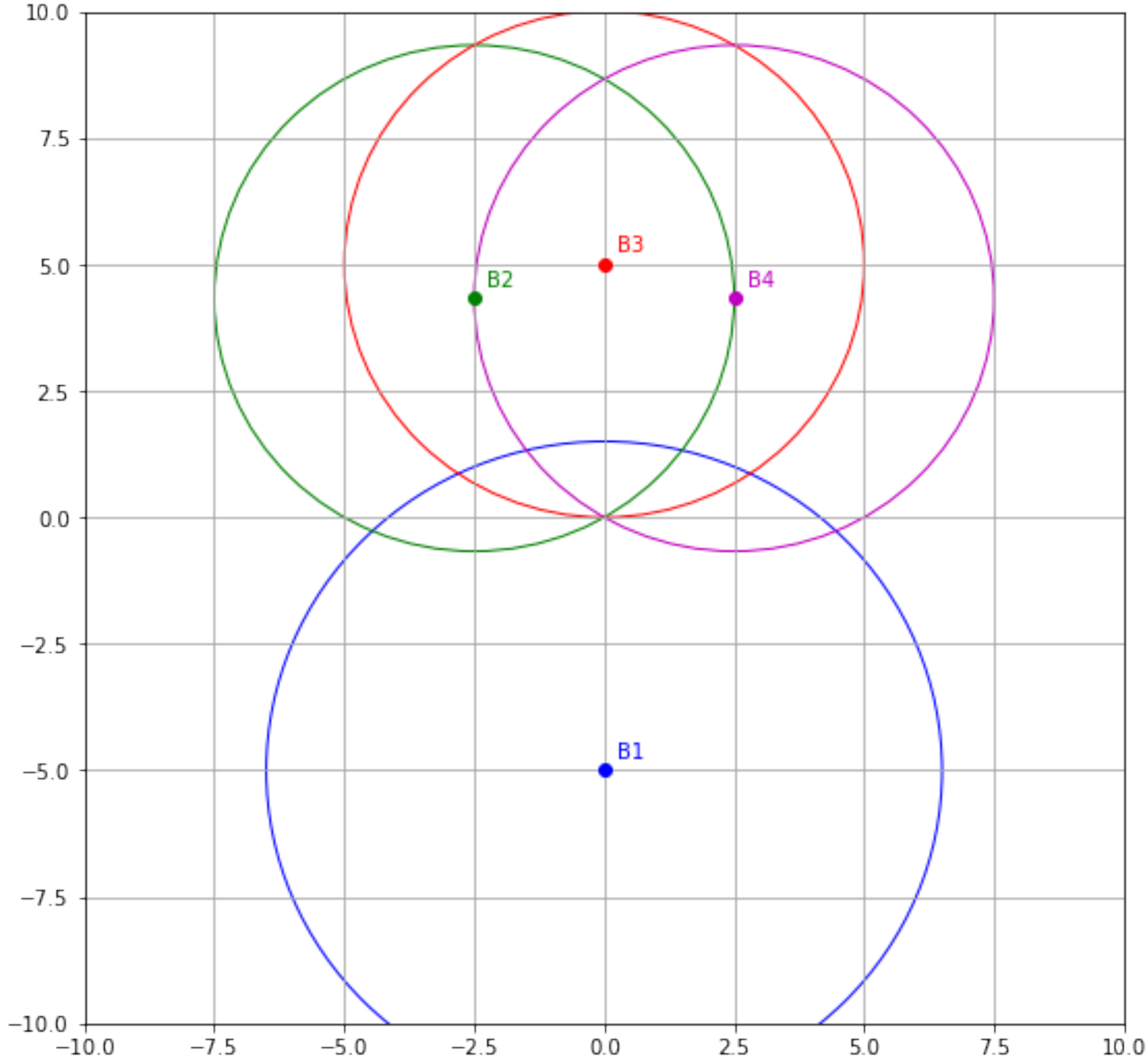
In [17]:
```
plot(ideal_distances)
```



In [18]:
```
plot(imperfect_distances)
```



In [19]:
```
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 [60]:

```
A = np.zeros((3,2))

A[0][0] = 2 * xpositions[0] - 2 * xpositions[1]
A[0][1] = 2 * ypositions[0] - 2 * ypositions[1]
A[1][0] = 2 * xpositions[0] - 2 * xpositions[2]
A[1][1] = 2 * ypositions[0] - 2 * ypositions[2]
A[2][0] = 2 * xpositions[0] - 2 * xpositions[3]
A[2][1] = 2 * ypositions[0] - 2 * 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 [61]:
```
def make_b(distances):
    """
    Since `b` depends on `distances`, we implement it using a functior
    different `b` vectors depending of which set of distances we are i
    (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] = -(xpositions[1]**2 + ypositions[1]**2) + (xpositions[0]**2
    b[1] = -(xpositions[2]**2 + ypositions[2]**2) + (xpositions[0]**2
    b[2] = -(xpositions[3]**2 + ypositions[3]**2) + (xpositions[0]**2


    return b

make_b(ideal_distances)
```

Out[61]:
```
array([3.55271368e-15, 0.00000000e+00, 3.55271368e-15])
```

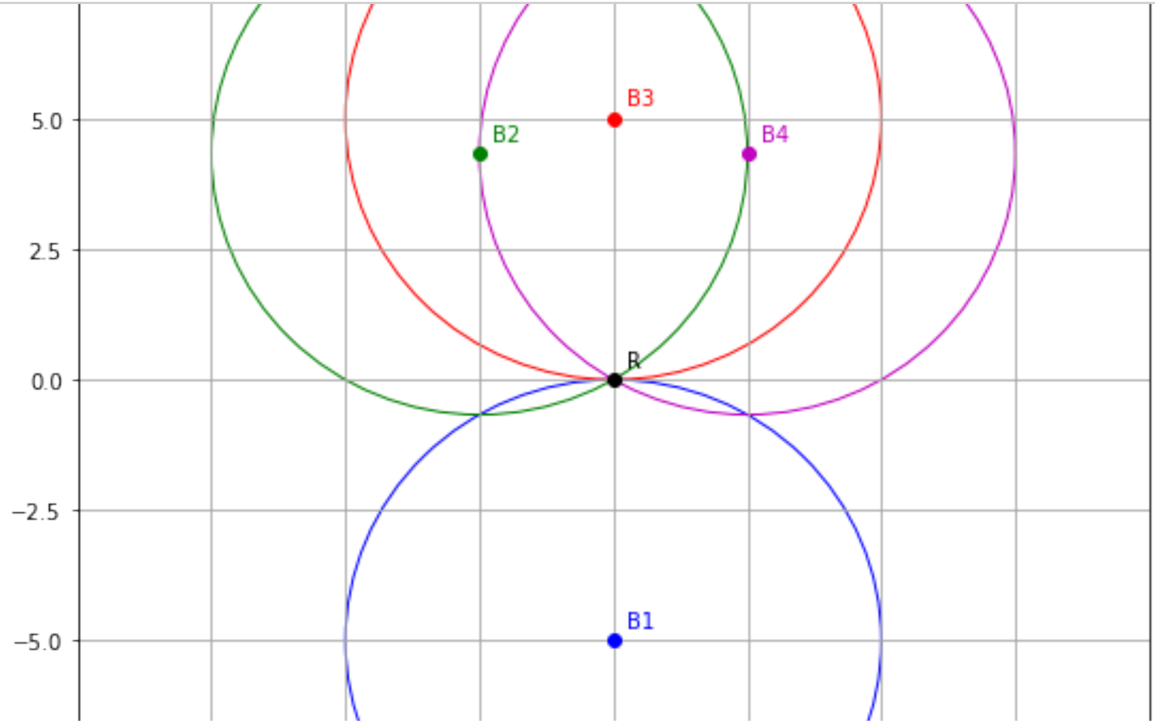
Part (g)

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 [62]:
```
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[62]:
```
array([ 8.59260618e-33, -1.20930181e-16])
```

In [63]:
```
plot(ideal_distances)
plot_point(estimate_position(ideal_distances))
```



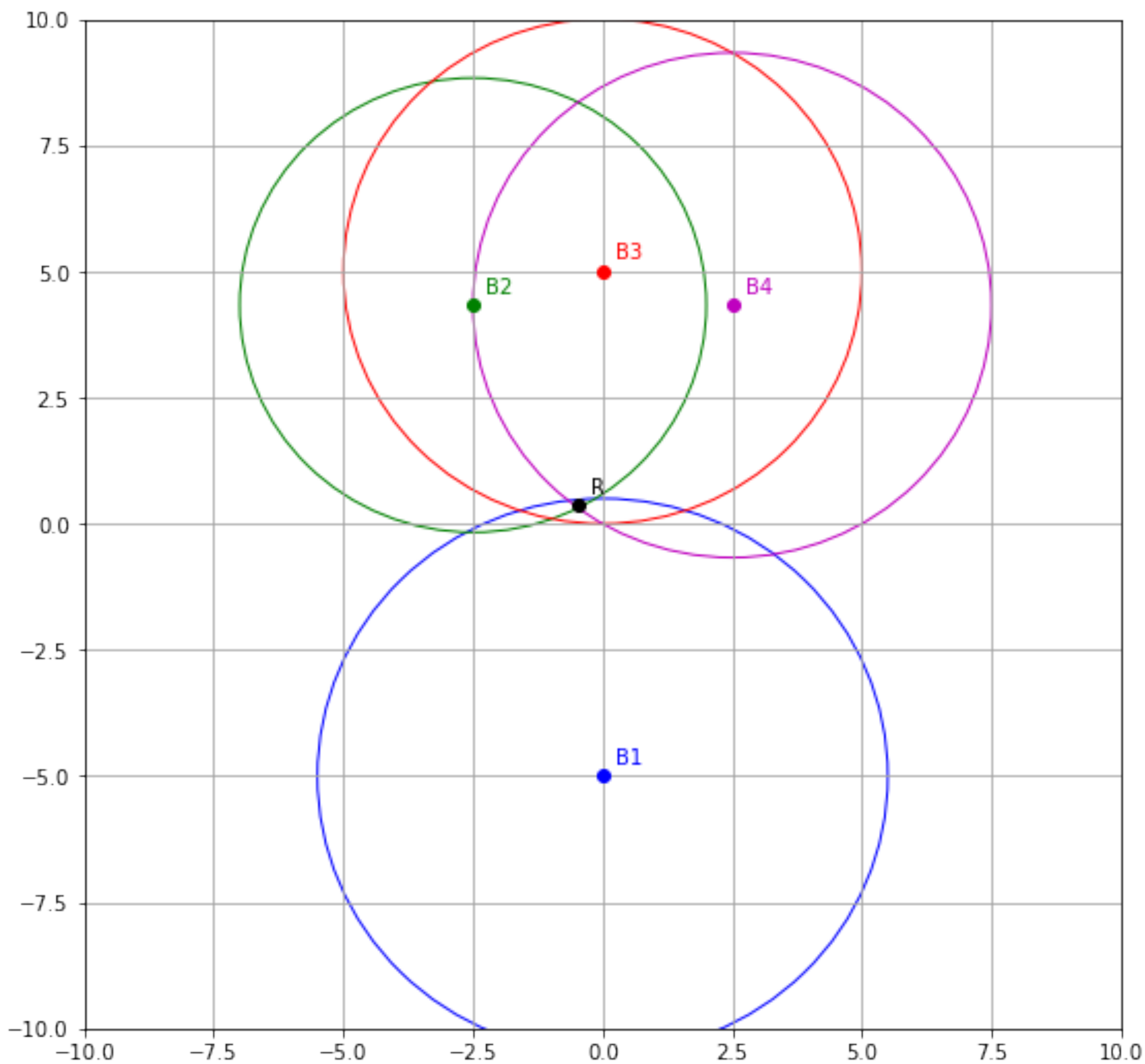
What about for the `imperfect_distances` and `one_bad_distances` ? Copy and modify the above code to compute and plot the least-squares trilateration solutions for those two cases, and comment on the quality of the solution in each case.

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 [64]:
```
### imperfect_distances case

def make_b(distances):
    """
    Since `b` depends on `distances`, we implement it using a function
    different `b` vectors depending of which set of distances we are i
    (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[1]**2 + ypo
```

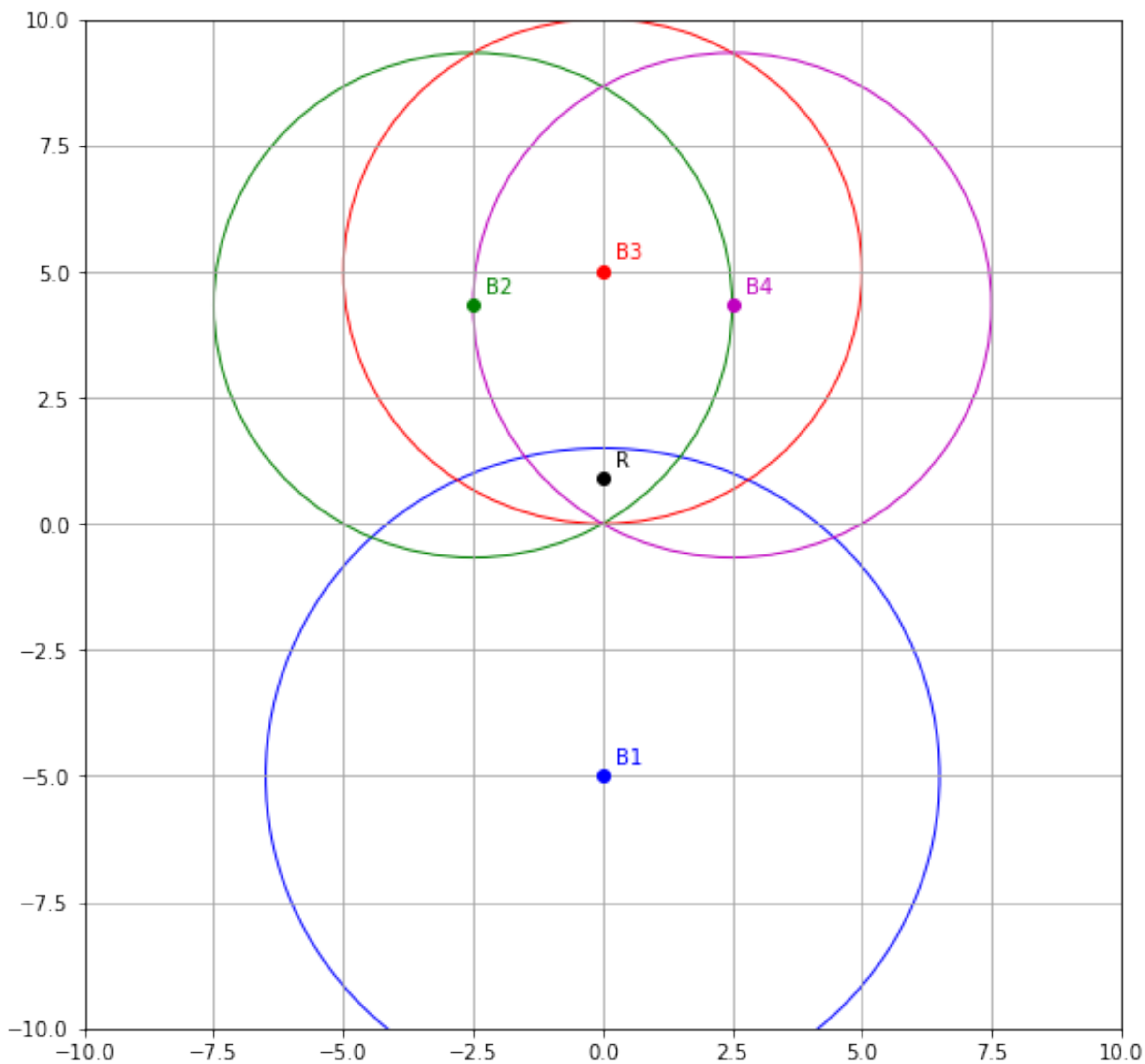
```
    b[1] = distances[2]**2 - distances[0]**2 - (xpositions[2]**2 + ypo
    b[2] = distances[3]**2 - distances[0]**2 - (xpositions[3]**2 + ypo


    return b

plot(imperfect_distances)
p = estimate_position(imperfect_distances)
print("point:", p)
plot_point(p)
```

```
point: [-0.475       0.35531308]
```



In [65]:
```
### one_bad_distances case

plot(one_bad_distances)
p = estimate_position(one_bad_distances)
print("point:", p)
plot_point(p)
```

```
point: [-6.40790822e-17  9.01832908e-01]
```



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:

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

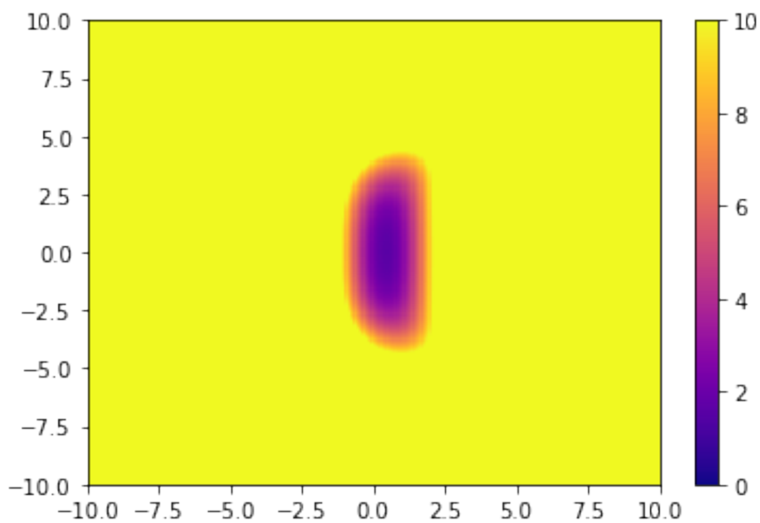
So we might want to minimize the following:

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

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 [66]:
```
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
    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 [70]:
```
with ideal distances:", cost_function([0,0], ideal_distances))
ted position with ideal distances:", cost_function(estimate_position(i
with imperfect distances:", cost_function([0,0], imperfect_distances))
ted position with imperfect distances:", cost_function(estimate_positi
with one bad distance:", cost_function([0,0], one_bad_distances))
ted position with one bad distance:", cost_function(estimate_position(
```

```
Cost of (0,0) with ideal distances: 0.0
Cost of estimated position with ideal distances: 0.0
Cost of (0,0) with imperfect distances: 0.5
Cost of estimated position with imperfect distances: 0.12767439181123
577
Cost of (0,0) with one bad distance: 2.25
Cost of estimated position with one bad distance: 2.31715040201415
```

Labeling Patients

In [37]:
```
import numpy as np
```

In [47]:
```
# Part B
A = np.load('gene_data_train.npy')
b = np.load('diabetes_train.npy')

model = list(np.linalg.lstsq(A, [i[0] for i in b], rcond=None)[0])
print(model)
```

```
[-0.15646168983619893, 0.09239417683325737, 0.4805397398225526, -0.58
47017953319975, -0.3535073412970997]
```

In [51]:
```
# 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')
correct = 0
for i in range(len(A_test)):
    data = A_test[i]
    pred = data[0] * model[0] + data[1] * model[1] + data[2] * model[2
    print("Prediction:", int(np.sign(pred)), "Actual:", b_test[i][0])

    if int(np.sign(pred)) == b_test[i][0]:
        correct += 1

print("Accuracy:", correct / len(b_test) * 100, "%")
```

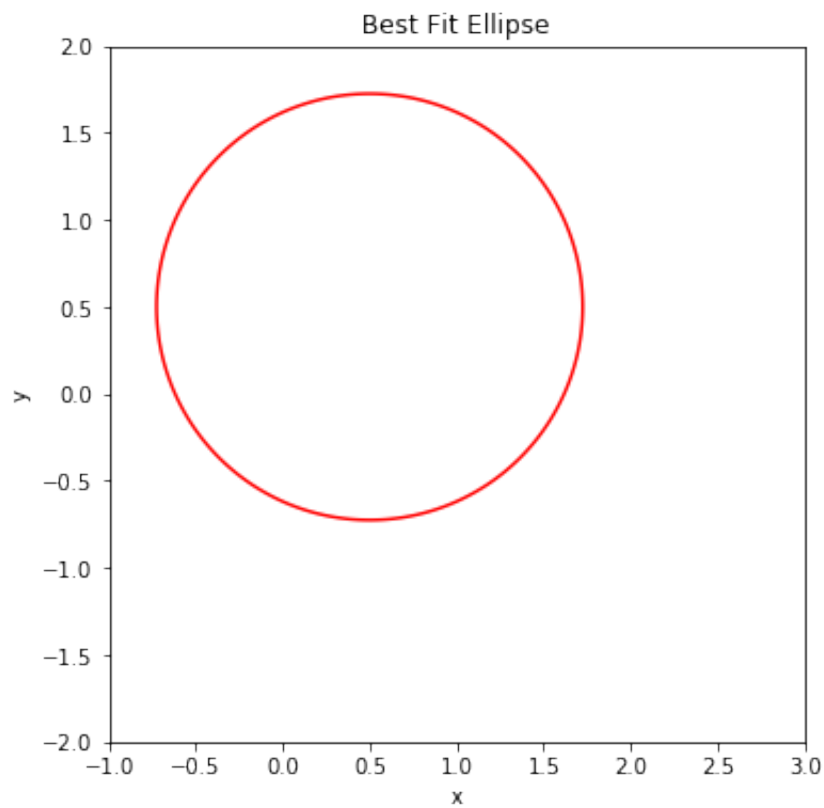
```
Prediction: 1 Actual: 1
Prediction: -1 Actual: -1
Prediction: -1 Actual: -1
Prediction: 1 Actual: 1
Accuracy: 100.0 %
```

Image Analysis

In [71]:
```
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 [72]:
```
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 [73]:
```
# 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)
```

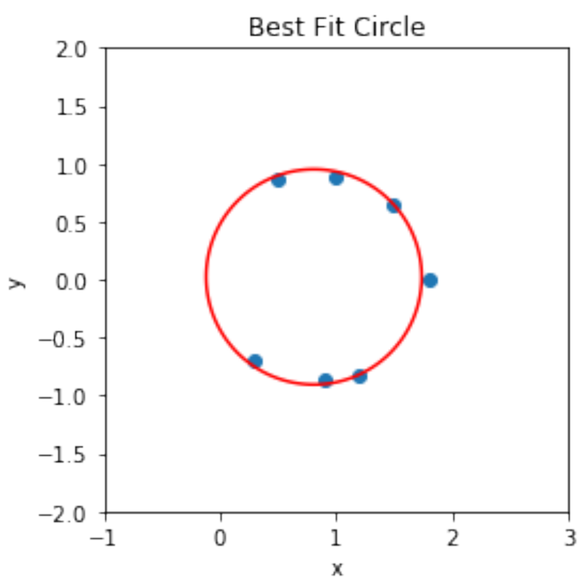


You may find plt.scatter (http://matplotlib.org/api/pyplot_api.html) useful for plotting the points.

In [85]:

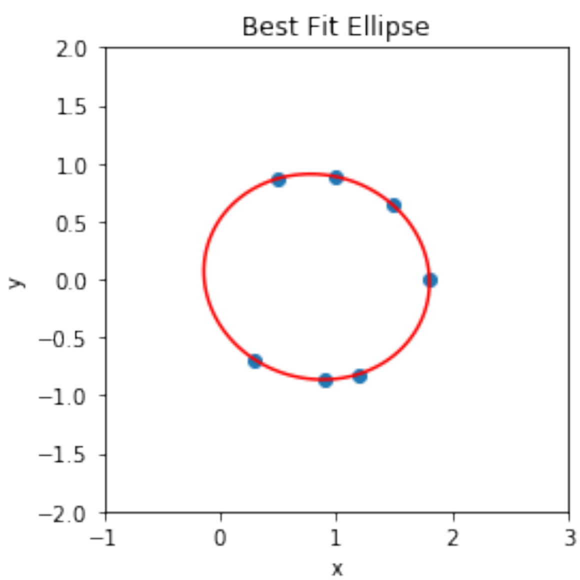
```
# PART C

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 = [1] * 7

circle_params = np.linalg.lstsq(A, b, rcond=None)[0]
print(circle_params)
plot_circle(circle_params[0],circle_params[1],circle_params[2])

# ||e|| / N
print("||e|| / N:",np.linalg.norm(np.dot(A, circle_params) - b) / 7)
```

```
[ 4.87314137 -7.89293482 -0.22651484]
||e|| / N: 0.1374905622431481
```



In [86]:
```
# PART D

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

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

ellipse_params = np.linalg.lstsq(A, b, rcond=None)[0]
print(ellipse_params)
plot_ellipse(ellipse_params[0],ellipse_params[1],ellipse_params[2],ell

# ||e|| / N
print("||e|| / N:",np.linalg.norm(np.dot(A, ellipse_params) - b) / 7)
```

```
[ 4.10382951  0.48711384  4.93938449 -6.85032284 -0.62259259]
||e|| / N: 0.012853829087236146
```



In [ ]: