NMEP HW #1

1. a. $\begin{bmatrix}1\\2\end{bmatrix} + \begin{bmatrix}3\\2\end{bmatrix} = \boxed{\begin{bmatrix}4\\4\end{bmatrix}}$

b. $\begin{bmatrix}1\\2\\3\end{bmatrix} + \begin{bmatrix}3\\2\\1\end{bmatrix} + \begin{bmatrix}4\\2\\0\end{bmatrix} = \boxed{\begin{bmatrix}8\\6\\4\end{bmatrix}}$

c. $\begin{bmatrix}0&1\\1&0\end{bmatrix}\begin{bmatrix}1\\2\end{bmatrix} = \boxed{\begin{bmatrix}2\\1\end{bmatrix}}$

2. a. Nullspace: $A\vec{x} = \vec{0}$

$$\left[\begin{array}{ccc|c}1&2&3&0\\-1&4&2&0\\0&6&5&0\end{array}\right]$$

$R_2 = R_1 + R_2$ $\left[\begin{array}{ccc|c}1&2&3&0\\0&6&5&0\\0&6&5&0\end{array}\right]$

$R_3 = R_2 - R_3$ $\left[\begin{array}{ccc|c}1&2&3&0\\0&6&5&0\\0&0&0&0\end{array}\right]$

$R_1 = R_1 - \frac{1}{3}R_2$ $\left[\begin{array}{ccc|c}1&0&\frac{4}{3}&0\\0&6&5&0\\0&0&0&0\end{array}\right]$

$R_2 = \frac{R_2}{6}$ $\left[\begin{array}{ccc|c}1&0&4/3&0\\0&1&5/6&0\\0&0&0&0\end{array}\right]$

$x_1 + \frac{4}{3}x_3 = 0 \rightarrow x_1 = -\frac{4}{3}x_3$

$x_2 + \frac{5}{6}x^3 = 0 \rightarrow x_2 = -\frac{5}{6}x_3$

$$\vec{x} = \begin{bmatrix}-\frac{4}{3}\\-\frac{5}{6}\\1\end{bmatrix} x_3 = \boxed{\text{span}\left\{\begin{bmatrix}-8\\-5\\6\end{bmatrix}\right\}}$$

Columnspace: Only 2 of the vectors are linearly independent, so column space is

$$\boxed{\text{span}\left\{\begin{bmatrix}-1\\0\end{bmatrix}, \begin{bmatrix}2\\4\\6\end{bmatrix}\right\}}$$

b. **No**, a linear transformation can be represented by a matrix. But for 2 matrices, AB is not necessarily equal to BA. For example, if $A = \begin{bmatrix}1&2\\3&4\end{bmatrix}$ and $B = \begin{bmatrix}5&6\\7&8\end{bmatrix}$,

2. b (cont.). $AB = \begin{bmatrix}1&2\\3&4\end{bmatrix}\begin{bmatrix}5&6\\7&8\end{bmatrix} = \begin{bmatrix}19&22\\43&50\end{bmatrix}$

$BA = \begin{bmatrix}5&6\\7&8\end{bmatrix}\begin{bmatrix}1&2\\3&4\end{bmatrix} = \begin{bmatrix}23&34\\31&46\end{bmatrix}$. } $BA \neq AB$.

Therefore, $T_1(T_2(\vec{v})) = T_2(T_1(\vec{v}))$ is not always correct.

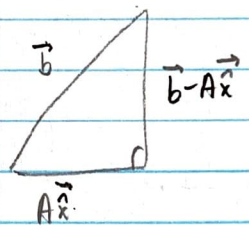
c. No, take this counter example. If $T_1 = \begin{bmatrix}1&0\\0&0\end{bmatrix}$ and $T_2 = \begin{bmatrix}0&0\\0&1\end{bmatrix}$,

Then $T_1(T_2(\vec{v}))$

$= \begin{bmatrix}1&0\\0&0\end{bmatrix}\begin{bmatrix}0&0\\0&1\end{bmatrix}\vec{v}$

$= \begin{bmatrix}0&0\\0&0\end{bmatrix}\vec{v}$

$= \vec{0}$, but T_1 and T_2 both aren't all-zero matrices.

3. 

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

$= \left(\begin{bmatrix}1&-1&1\\2&4&2\\0&6&0\end{bmatrix}\begin{bmatrix}1&2&0\\-1&4&6\\1&2&0\end{bmatrix}\right)^{-1}\begin{bmatrix}1&-1&1\\2&4&2\\0&6&0\end{bmatrix}\begin{bmatrix}3\\-1\\5\end{bmatrix}$

$= \begin{bmatrix}3&0&-6\\0&24&24\\-6&24&36\end{bmatrix}^{-1}\begin{bmatrix}9\\12\\-6\end{bmatrix}$

$\det(m) = -0\begin{vmatrix}0&24\\-6&36\end{vmatrix} + 24\begin{vmatrix}3&-6\\-6&36\end{vmatrix} - 24\begin{vmatrix}3&-6\\0&24\end{vmatrix}$

$= 24(108-36) - 24(72-0) = 0 \rightarrow$ not invertible

Since A does not have linearly independent columns ($-2\begin{bmatrix}1\\-1\\1\end{bmatrix} + \begin{bmatrix}2\\4\\2\end{bmatrix} = \begin{bmatrix}0\\6\\0\end{bmatrix}$)

let $A = \begin{bmatrix}1&2\\-1&4\\1&2\end{bmatrix}$.

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

$= \left(\begin{bmatrix}1&-1&1\\2&4&2\end{bmatrix}\begin{bmatrix}1&2\\-1&4\\1&2\end{bmatrix}\right)^{-1}\begin{bmatrix}1&-1&1\\2&4&2\end{bmatrix}\begin{bmatrix}3\\-1\\5\end{bmatrix}$

3. a (cont'd). $= \begin{bmatrix} 3 & 0 \\ 0 & 24 \end{bmatrix}^{-1} \begin{bmatrix} 9 \\ 12 \end{bmatrix}$

$= \frac{1}{72} \begin{bmatrix} 24 & 0 \\ 0 & 3 \end{bmatrix} \begin{bmatrix} 9 \\ 12 \end{bmatrix}$

$= \begin{bmatrix} \frac{1}{3} & 0 \\ 0 & \frac{1}{24} \end{bmatrix} \begin{bmatrix} 9 \\ 12 \end{bmatrix} = \begin{bmatrix} 3 \\ 0.5 \end{bmatrix}, \rightarrow \hat{\vec{x}} = \boxed{\begin{bmatrix} 3 \\ 0.5 \\ 0 \end{bmatrix}}$

b. $A\hat{\vec{x}} = \begin{bmatrix} 1 & 2 \\ -1 & 4 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ 0.5 \end{bmatrix} = \boxed{\begin{bmatrix} 4 \\ -1 \\ 4 \end{bmatrix}}$

c. $\| \vec{b} - A\hat{\vec{x}} \| = \left\| \begin{bmatrix} 3 \\ -1 \\ 5 \end{bmatrix} - \begin{bmatrix} 4 \\ -1 \\ 4 \end{bmatrix} \right\| = \left\| \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} \right\| = \sqrt{(-1)^2 + 0^2 + 1^2} = \boxed{\sqrt{2}}$

4. $\min_w \| X\vec{w} - \vec{y} \|_2^2 + \lambda \| w \|_2^2$

$= \min_w (X\vec{w} - \vec{y})^T (X\vec{w} - \vec{y}) + \lambda \vec{w}^T \vec{w}$

$= \min_w \vec{y}^T\vec{y} + \vec{w}^T X^T X \vec{w} - \vec{y}^T X\vec{w} - \vec{w}^T X^T \vec{y} + \lambda \vec{w}^T I \vec{w}$

$\nabla_{\vec{w}} = (X^TX + (X^TX)^T)\vec{w} - 2X^T\vec{y} + \lambda(I + I^T)\vec{w}$

$= (X^TX + X^TX)\vec{w} - 2X^T\vec{y} + \lambda(2I)\vec{w}$

$= 2X^TX\vec{w} - 2X^T\vec{y} + 2\lambda I\vec{w} = 0$

$(X^TX + \lambda I)\vec{w} = X^T\vec{y}$

$\boxed{\vec{w} = (X^TX + \lambda I)^{-1} X^T\vec{y}}$

We can tune the hyperparameter λ by trying out a bunch of different values and then checking how well the corresponding $\vec{w}$ fits our data (GridSearch).

Intro to Numpy

What is Numpy and why do we use it?

It's an awesome python package that adds support for large, multi-dimensional arrays. Really good for vector operations, matrix operations because its super parallelized so its super fast!

Why not Python arrays?

Python arrays has certain limitations: they don't support "vectorized" operations like elementwise addition and multiplication, and the fact that they can contain objects of differing types mean that Python must store type information for every element, and must execute type dispatching code when operating on each element. This also means that very few list operations can be carried out by efficient C loops – each iteration would require type checks and other Python API bookkeeping.

Importing numpy

Functions for numerical computiing are provided by a separate module called `numpy` which we must import.

By convention, we import numpy using the alias `np` .

Once we have done this we can prefix the functions in the numpy library using the prefix `np` .

In [1]:
```
# This is the de facto way to import NumPy. You probably don't want to
write numpy.whatever every time
import numpy as np
```

Numpy Arrays

NumPy arrays are the workhorse of the library. A NumPy array is essentially a bunch of data coupled with some metadata:

type: the type of objects in the array. This will typically be floating-point numbers for our purposes, but other types can be stored. The type of an array can be accessed via the `dtype` attribute.

shape: the dimensions of the array. This is given as a tuple, where element i of the tuple tells you how the "length" of the array in the ith dimension. For example, a 10-dimensional vector would have shape (10,), a 32-by-100 matrix would have shape (32,100), etc. The shape of an array can be accessed via the `shape` attribute.

Let's see some examples! There are number of ways to construct arrays. One is to pass in a Python sequence (such as list or tuple) to the `np.array` function:

In [2]:
```
np.array([1, 2.3, -6])
```

Out[2]:
```
array([ 1. ,  2.3, -6. ])
```

We can also easily create ordered numerical lists!

In [3]:
```
# Remember we zero index so you will actually get 0 to 6!
print(np.arange(7))
# Remember the list wont include 9
print(np.arange(3, 9))
```

```
[0 1 2 3 4 5 6]
[3 4 5 6 7 8]
```

We can also customize these list with a third paramter that specifices step size

In [4]:
```
np.arange(0.0, 100.0, 10.0)
```

Out[4]:
```
array([ 0., 10., 20., 30., 40., 50., 60., 70., 80., 90.])
```

To create a multi-dimensional array, we'll need to nest the sequences:

In [5]:
```
np.array([[1, 2.3, -6], [7, 8, 9]])
```

Out[5]:
```
array([[ 1. ,  2.3, -6. ],
       [ 7. ,  8. ,  9. ]])
```

Neat!

There are also many convenience functions for constructing special arrays. Here are some that might be useful:

In [6]:
```
# The identity matrix of given size
np.eye(7)
```

Out[6]:
```
array([[1., 0., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 0., 0., 1.]])
```

In [7]:
```
# A matrix with the given vector on the diagonal
np.diag([1.1,2.2,3.3])
```

Out[7]:
```
array([[1.1, 0. , 0. ],
       [0. , 2.2, 0. ],
       [0. , 0. , 3.3]])
```

In [8]:
```
#An array of all zeros or ones with the given shape
np.zeros((8,4)), np.ones(3)
```

Out[8]:
```
(array([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]]), array([1., 1., 1.]))
```

In [9]:
```
# An array with a given shape full of a specified value
np.full((3,4), 2.1)
```

Out[9]:
```
array([[2.1, 2.1, 2.1, 2.1],
       [2.1, 2.1, 2.1, 2.1],
       [2.1, 2.1, 2.1, 2.1]])
```

In [10]:
```
# A random (standard normal distribution) array with the given shape
np.random.randn(5,6)
```

Out[10]:
```
array([[-0.21012433, -0.395064  ,  0.58699609,  0.21996799, -0.63239
225,
         0.26370822],
       [-0.46016365, -1.18521576, -0.40156712, -0.01302791,  1.16057
806,
         0.75917215],
       [-0.1697336 ,  1.30926073,  0.68306209,  0.05048018, -0.25175
731,
         1.06677987],
       [-0.42961748, -0.34984805, -0.7626864 , -0.65096765,  0.66512
512,
        -0.2647765 ],
       [ 0.10989286,  0.4199001 ,  0.28404666, -0.66111752,  1.26021
329,
         0.25438349]])
```

Okay your turn! In the cell belows try and create:

A diagonal matrix with values from 1-20 (try and create this and only type two numbers!)

In [11]:
```
#Your answer here
np.diag(np.arange(1,21))
```

Out[11]:
```
array([[ 1,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0,
         0,  0,  0,  0],
       [ 0,  2,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0,
         0,  0,  0,  0],
       [ 0,  0,  3,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0,
         0,  0,  0,  0],
       [ 0,  0,  0,  4,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0,
         0,  0,  0,  0],
       [ 0,  0,  0,  0,  5,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0,
         0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  6,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0,
         0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  7,  0,  0,  0,  0,  0,  0,  0,  0,
0,
         0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  8,  0,  0,  0,  0,  0,  0,  0,
```

```
0,
          0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  9,  0,  0,  0,  0,  0,  0,
0,
          0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0, 10,  0,  0,  0,  0,  0,
0,
          0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 11,  0,  0,  0,  0,
0,
          0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 12,  0,  0,  0,
0,
          0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 13,  0,  0,
0,
          0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 14,  0,
0,
          0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0, 15,
0,
          0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
16,
          0,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0,
         17,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0,
          0, 18,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0,
          0,  0, 19,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,  0,
0,
          0,  0,  0, 20]])
```

Okay now let's suppose we have some data in an array so we can start doing stuff with it.

In [12]:
```
A = np.random.randn(10,5); x = np.random.randn(5)
A
```

Out[12]:
```
array([[-2.92468280e-01,  8.79546762e-01, -1.31717614e+00,
         1.08877423e+00, -1.13145551e+00],
       [ 1.32223145e+00,  6.74591595e-01, -6.81043247e-01,
        -7.36051389e-01, -8.34278291e-02],
       [-4.68542890e-01,  4.42117298e-01,  1.21137837e+00,
        -1.33352354e+00, -1.38875897e+00],
       [ 2.02622100e-01, -5.24428790e-01, -1.04213330e+00,
         3.22529310e-01, -9.88326861e-01],
       [-1.20211711e-03,  1.02885550e+00,  5.90903166e-01,
        -6.08259125e-01, -1.58422186e+00],
       [-3.67900439e-01,  1.46486428e+00,  9.47578215e-01,
        -8.01376683e-01,  1.64676644e-01],
       [-4.11347485e-01, -8.57052284e-01, -1.46004739e+00,
        -3.91742223e-01,  6.33541639e-02],
       [-2.27954001e-01, -6.15088897e-01,  8.01566650e-01,
         7.83904053e-01,  4.73235355e-01],
       [-7.31756484e-01,  1.52676976e-01,  1.08942979e+00,
        -4.37079083e-01, -5.59649993e-01],
       [-1.63605277e+00, -3.19578330e-01, -3.16311099e-01,
        -2.49333922e+00,  7.77339143e-02]])
```

One useful thing that NumPy lets us do efficiently is apply the same function to every element in an array. You'll often need to e.g. exponentiate a bunch of values, but if you use a list comprehension or map with the builtin Python math functions it may be really slow. Instead just write

In [13]:
```
# log, sin, cos, etc. work similarly
np.exp(A)
```

Out[13]:
```
array([[0.74641892, 2.40980724, 0.26789072, 2.97063053, 0.32256342],
       [3.75178395, 1.96323102, 0.50608874, 0.47900158, 0.91995748],
       [0.62591363, 1.55599825, 3.3581102 , 0.263547  , 0.24938461],
       [1.2246096 , 0.59189336, 0.35270146, 1.38061536, 0.37219891],
       [0.99879861, 2.79786185, 1.80561845, 0.5442976 , 0.20510733],
       [0.69218609, 4.32695593, 2.5794552 , 0.44871081, 1.17901182],
       [0.66275659, 0.42441128, 0.23222527, 0.67587832, 1.0654041 ],
       [0.79616088, 0.54059284, 2.22903031, 2.1900055 , 1.60517913],
       [0.48106327, 1.16494861, 2.97257859, 0.64592035, 0.57140903],
       [0.19474724, 0.7264553 , 0.72883268, 0.08263357, 1.08083503]]
)
```

We can take the sum/mean/standard deviation/etc. of all the elements in an array:

In [14]:
```
np.sum(x), np.mean(x), np.std(x)
```

Out[14]:
```
(-0.762267691287962, -0.1524535382575924, 0.4153529425783442)
```

You can also specify an axis over which to compute the sum if you want a vector of row/column sums (again, sum here can be replaced with mean or other operations):

In [15]:
```
# Create an array with numbers in the range 0,...,3 (similar to the no
rmal Python range function,
# but it returns a NumPy array) and then reshape it to a 2x2 matrix
B = np.arange(4).reshape((2,2))

# Original matrix, column sum, row sum
B, np.sum(B, axis=0), np.sum(B, axis=1)
```

Out[15]:
```
(array([[0, 1],
        [2, 3]]), array([2, 4]), array([1, 5]))
```

Linear Algebra

By now we have a pretty good idea of how data is stored and accessed within NumPy arrays. But we typically want to do something more "interesting", which for our ML purposes usually means linear algebra operations. Fortunately, numpy has good support for such routines. Let's see some examples!

In [16]:
```
# Matrix-vector product. The dimensions have to match, of course
A.dot(x)
# Note that in Python3 there is also a slick notation A @ x which does
the same thing
```

Out[16]:
```
array([ 0.99288794, -1.16042446, -0.21681476,  0.92161511, -0.083960
16,
       -1.17795819,  0.34539557,  0.57268078,  0.14905389, -0.825641
11])
```

In [17]:
```
# Transpose a matrix
A.T
```

Out[17]:
```
array([[-2.92468280e-01,  1.32223145e+00, -4.68542890e-01,
         2.02622100e-01, -1.20211711e-03, -3.67900439e-01,
        -4.11347485e-01, -2.27954001e-01, -7.31756484e-01,
        -1.63605277e+00],
       [ 8.79546762e-01,  6.74591595e-01,  4.42117298e-01,
        -5.24428790e-01,  1.02885550e+00,  1.46486428e+00,
        -8.57052284e-01, -6.15088897e-01,  1.52676976e-01,
        -3.19578330e-01],
       [-1.31717614e+00, -6.81043247e-01,  1.21137837e+00,
        -1.04213330e+00,  5.90903166e-01,  9.47578215e-01,
        -1.46004739e+00,  8.01566650e-01,  1.08942979e+00,
        -3.16311099e-01],
       [ 1.08877423e+00, -7.36051389e-01, -1.33352354e+00,
         3.22529310e-01, -6.08259125e-01, -8.01376683e-01,
        -3.91742223e-01,  7.83904053e-01, -4.37079083e-01,
        -2.49333922e+00],
       [-1.13145551e+00, -8.34278291e-02, -1.38875897e+00,
        -9.88326861e-01, -1.58422186e+00,  1.64676644e-01,
         6.33541639e-02,  4.73235355e-01, -5.59649993e-01,
         7.77339143e-02]])
```

Now that you're familiar with numpy feel free to check out the documentation and see what else you can do! Documentation can be found here: https://docs.scipy.org/doc/ (https://docs.scipy.org/doc/)

Exercises

Lets try out all the new numpy stuff we just learned! Even if you have experience in numpy we suggest trying these out.

1) Create a vector of size 10 containing zeros

In [18]:
```
## FILL IN YOUR ANSWER HERE ##
v = np.zeros(10)
v
```

Out[18]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

2) Now change the fith value to be 5

In [20]:
```
## FILL IN YOUR ANSWER HERE ##
v[4] = 5
v
```

Out[20]:
```
array([0., 0., 0., 0., 5., 0., 0., 0., 0., 0.])
```

3) Create a vector with values ranging from 10 to 49

In [22]:
```
## FILL IN YOUR ANSWER HERE ##
v = np.arange(10, 50)
v
```

Out[22]:
```
array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 2
5, 26,
       27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 4
2, 43,
       44, 45, 46, 47, 48, 49])
```

4) Reverse the previous vector (first element becomes last)

In [23]:
```
## FILL IN YOUR ANSWER HERE ##
v = v[::-1]
v
```

Out[23]:
```
array([49, 48, 47, 46, 45, 44, 43, 42, 41, 40, 39, 38, 37, 36, 35, 3
4, 33,
       32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 1
7, 16,
       15, 14, 13, 12, 11, 10])
```

5) Create a 3x3 matrix with values ranging from 0 to 8. Create a 1D array first and then resshape it. </b<

In [24]:
```
## FILL IN YOUR ANSWER HERE ##
np.arange(9).reshape((3,3))
```

Out[24]:
```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

6) Create a 3x3x3 array with random values

In [25]:
```
## FILL IN YOUR ANSWER HERE ##
np.random.randn(3,3,3)
```

Out[25]:
```
array([[[ 0.10553989, -1.44918101,  1.76692187],
        [ 0.9600631 , -2.12465232,  0.07983749],
        [ 0.88609752,  0.47117387, -0.55992636]],

       [[ 1.09676838,  1.59762001,  1.27709261],
        [ 0.75616915, -1.07806943,  0.18685633],
        [-0.62523703, -1.09455755, -0.23230448]],

       [[ 0.09512103, -3.39568338,  1.290432  ],
        [-0.97324084,  0.99904464,  0.37482322],
        [-0.84228701, -1.41113008, -0.81834712]]])
```

7) Create a random array and find the sum, mean, and standard deviation

In [26]:
```
## FILL IN YOUR ANSWER HERE ##
v = np.random.randn(100)
np.sum(v), np.mean(v), np.std(v)
```

Out[26]:
```
(0.20254356081693747, 0.0020254356081693746, 1.0827826632876085)
```

In [ ]:
```

```

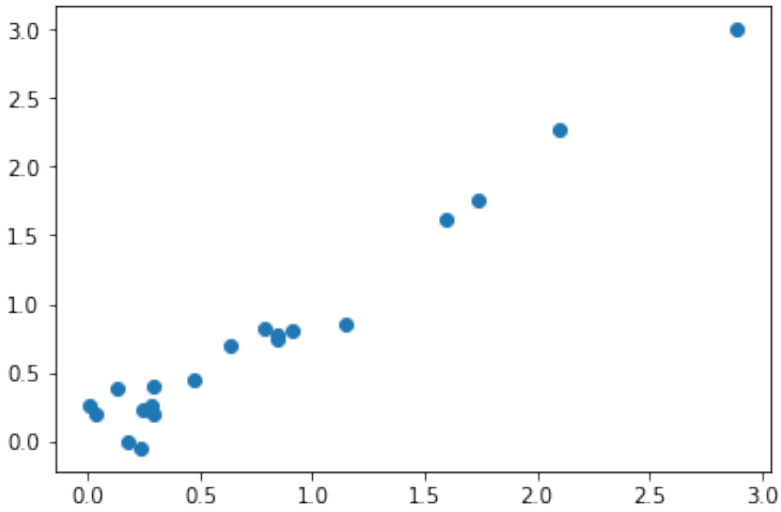
Linear Regression - Gaussian Orbits

Imports needed in this notebook: `numpy` (as np), `matplotlib.pyplot` (as plt), from sklearn: `LinearRegression`, `ElasticNet`, and `mean_squared_error`. Search up documentation if you have issues with any of the scikit learn things. This first part will be a warmup with numpy, linear regression, and scikit learn.

In [3]:
```
### YOUR IMPORT CODE HERE
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression, ElasticNet
from sklearn.metrics import mean_squared_error
```

First, we'll generate some random data and find the line of best fit.

In [7]:
```
num_data = 20

def gen_data(n):

    x = []
    y = []

    for i in range(n):
        rand = abs(np.random.randn())
        x += [rand]
        y += [.15 * np.random.randn() + rand]

    plt.scatter(x, y)
    plt.show()

    return (x, y)

data = gen_data(num_data)
data = [(data[0][i], data[1][i]) for i in range(num_data)]
data
```

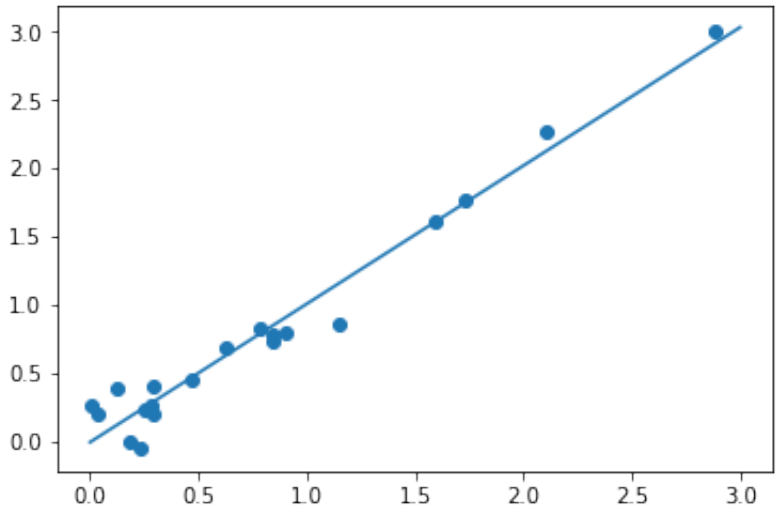


Out[7]:
```
[(0.8463782847314022, 0.7362085678615494),
 (0.7870448153101179, 0.8252617577179918),
 (0.12874811072226117, 0.388945908839599),
 (0.8441908955035079, 0.7811857947591122),
 (0.277997439039963, 0.26757260272311256),
 (1.149828611799792, 0.8579499967276797),
 (0.9062900503387852, 0.8001301680305749),
 (0.46880128078087707, 0.45253152660085294),
 (1.5940586932725254, 1.6134183106313167),
 (0.009976240390721394, 0.2670116990253635),
 (1.7355415858677445, 1.7624280826362329),
 (0.6315276383656515, 0.6909427154520842),
 (0.294433815590066, 0.20399015461893544),
 (2.886809868918759, 3.0012619737982424),
 (0.18209952293050177, -0.002718136538349153),
 (2.1008826641350455, 2.2749239682811884),
 (0.2306681431508596, -0.05234509877551913),
 (0.24794407292381804, 0.2333180954596053),
 (0.29437782244312527, 0.4037965231652011),
 (0.034698550152521435, 0.19882754911689532)]
```

Numpy Linear Regression

Now, `data` is a list of tuples of data. Perform linear regression to find the line of best fit, using only numpy. Minimize squared loss. Use matplotlib to plot your line of best fit.

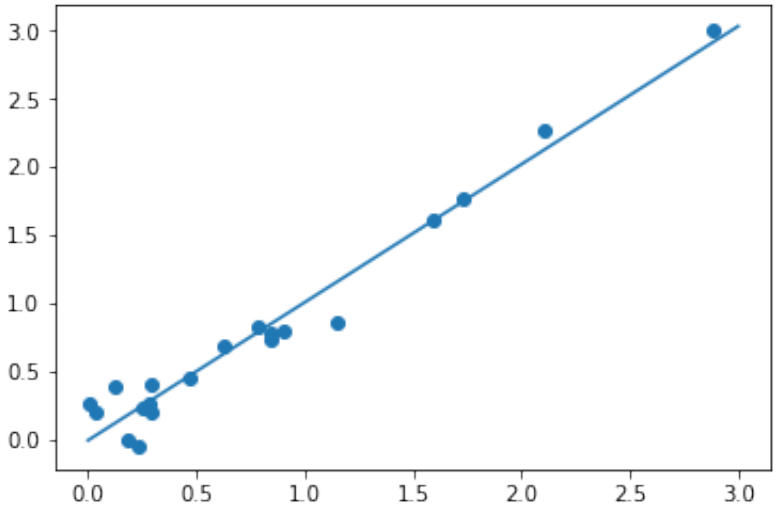
In [55]:
```
data1 = data[:] #.copy() was throwing an error; data1 is a copy of dat
a, dont modify data since we'll use it later
#YOUR CODE HERE
y = np.array([data1[i][1] for i in range(num_data)])
A = np.array([[data1[i][0], 1] for i in range(num_data)])
x_hat = np.dot(np.matmul(np.linalg.inv(np.matmul(A.T, A)), A.T), y)

plt.scatter(A[:,0], y)
line_x_data = np.linspace(0, 3, 1000)
plt.plot(line_x_data, x_hat[0] * line_x_data + x_hat[1])
plt.show()
```



Sklearn Linear Regression

Now, do the same thing, except using Scikit Learn. I recommend reading some documentation, specifically here: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html)

In [56]:
```
data2 = data[:] #.copy() was throwing an error
#YOUR CODE HERE
model = LinearRegression()
model.fit(A[:,0].reshape(-1,1), y)

plt.scatter(A[:,0], y)
line_x_data = np.linspace(0, 3, 1000)
plt.plot(line_x_data, model.coef_[0] * line_x_data + model.intercept_)
plt.show()
```



Gaussian Orbits

In this homework, we will use linear regression methods in order to determine the orbits of heavenly bodies.

Background

In 1801 the minor planet Ceres was first observed for a period of 40 days before moving behind the sun. To predict the location of Ceres astronomers would have to solve complicated non-linear differential equations, quite a task in an era before computers or calculators. However, Carl Friedrich Gauss had another idea. By single handedly developing the theory of least squares and linear regression and applying it to the problem of finding Ceres, Gauss managed to accurately predict the location of the minor planet nearly a year after it's last sighting.

In this problem we likewise attempt to predict the orbit of a "planet" and in the process "derive" the formula for an ellipse, the shape of orbits of heavenly bodies.

1. Generate Data

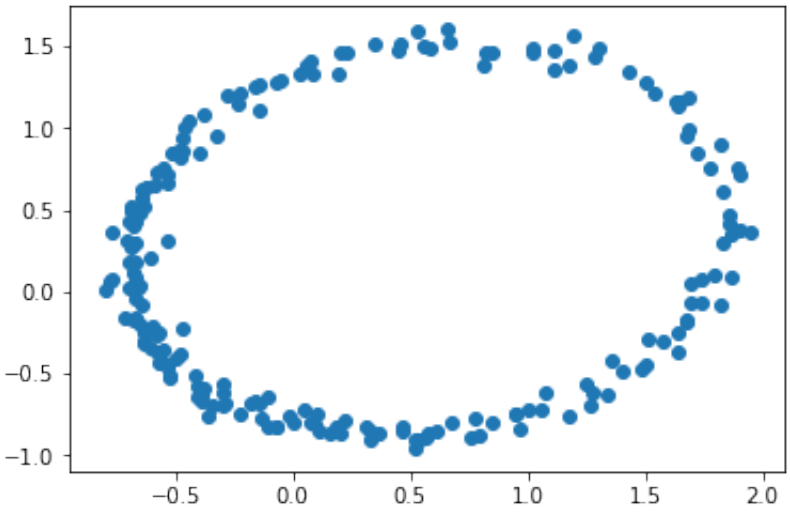
The idea here is we generate data in the shape of an ellipse. To do this we use the formula of an ellipse in polar coordinates:

$$r = \frac{ep}{1 - e\cos\theta}$$

where e is the eccentricity and p is the distance from the minor axis to the directrix (read "length"). In addition, we add random noise to the data.

We will then try to fit curves to our synthetic dataset.

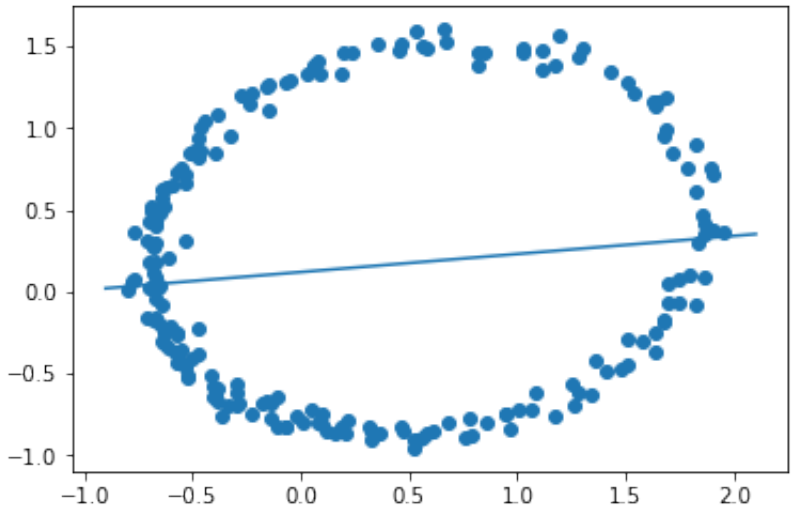
In [57]:
```
def gen_data(e, p, o):
    theta = np.linspace(0,2*np.pi, 200)

    # Ellipse with eccentricity e
    # Axis "length" p
    # Offset by .5 angularly
    r = e*p/(1-e*np.cos(theta - o))

    # transform to cartesian
    x = r * np.cos(theta)
    y = r * np.sin(theta)

    # Add noise
    x += np.random.randn(x.shape[0]) / 20
    y += np.random.randn(y.shape[0]) / 20

    # plot
    plt.scatter(x, y)
    plt.show()

    # saving
    np.save('x.npy', x)
    np.save('y.npy', y)
    return x, y

x, y = gen_data(.5, 2, .5)
```



2. Use sklearn's LinearRegression

Try to fit a `LinearRegression` model to `x` and `y` (let x be the independent variable and y be the dependent variable). Print out the `mean_squared_error` you get and plot both `x`, `y` (scatter plot), and the predicted orbit (line plot).

In [59]:
```
### YOUR CODE HERE
model = LinearRegression()
model.fit(x.reshape(-1, 1), y)

plt.scatter(x, y)
line_x_data = np.linspace(-0.9, 2.1, 1000)
plt.plot(line_x_data, model.coef_[0] * line_x_data + model.intercept_)
plt.show()

print("Mean squared error: " + str(mean_squared_error(y, model.coef_[0
] * x + model.intercept_)))
```




```
Mean squared error: 0.6559930382275464
```

This is not the best approach for our data. Please explain why below.

The data is not linear! In other words, the "line of best fit" is not a straight line in the Cartesian plane, but rather an ellipse.

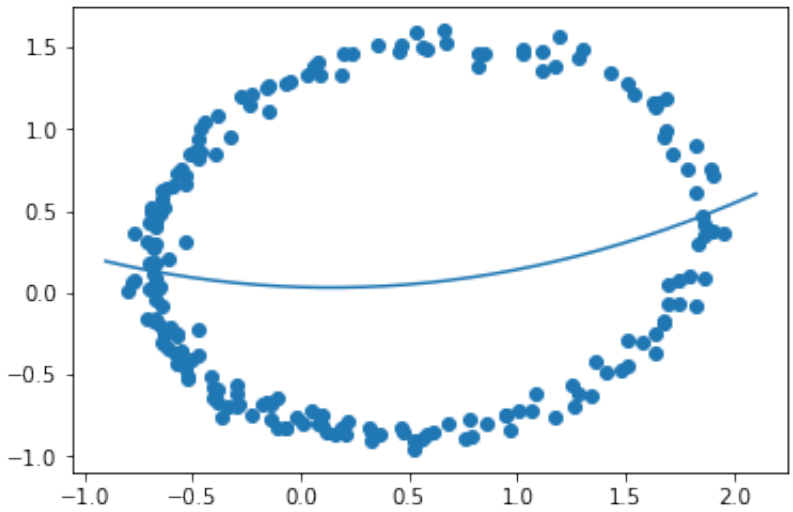
3. Experimentation time!

Try adding new features to your linear model by manipulating x! For example, try adding a quadratic term, x^2 or a root term like $\sqrt{x}$. Print out the MSE of your model and plot both `x`, `y` (scatter plot), and the predicted orbit (line plot). This time, your model should take in an expanded set of features and predict y.

Hint: `np.vstack` may be useful here.

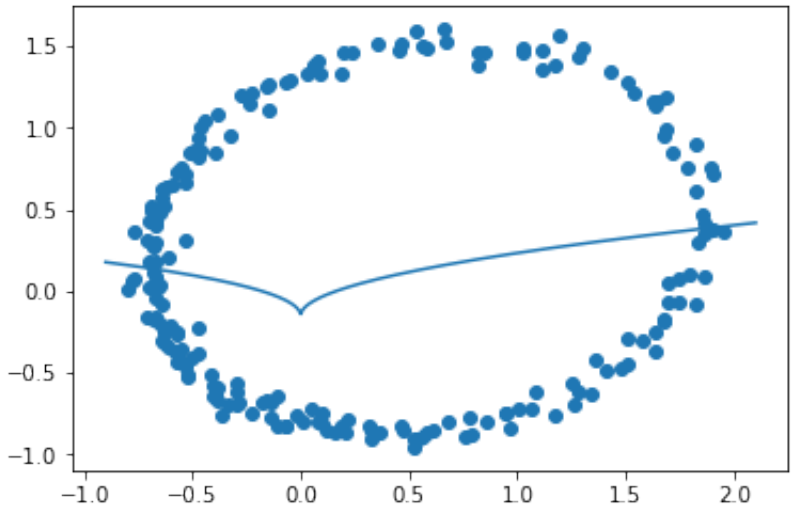
In [77]:
```
### YOUR CODE HERE
print("Adding x^2 term")
features = np.hstack((np.square(x).reshape(-1, 1), x.reshape(-1, 1)))
model = LinearRegression()
model.fit(features, y)

plt.scatter(x, y)
line_x_data = np.linspace(-0.9, 2.1, 1000)
line_y_data = np.dot(np.hstack((np.square(line_x_data).reshape(-1, 1),
line_x_data.reshape(-1, 1))), model.coef_) + model.intercept_
plt.plot(line_x_data, line_y_data)
plt.show()

y_pred = np.dot(features, model.coef_) + model.intercept_
print("Mean squared error: " + str(mean_squared_error(y, y_pred)))

print("\n\nAdding sqrt(|x|) term")
features = np.hstack((np.sqrt(np.abs(x)).reshape(-1, 1), x.reshape(-1,
1)))
model = LinearRegression()
#print(features)
model.fit(features, y)

plt.scatter(x, y)
line_x_data = np.linspace(-0.9, 2.1, 1000)
line_y_data = np.dot(np.hstack((np.sqrt(np.abs(line_x_data)).reshape(-
1, 1), line_x_data.reshape(-1, 1))), model.coef_) + model.intercept_
plt.plot(line_x_data, line_y_data)
plt.show()

y_pred = np.dot(features, model.coef_) + model.intercept_
print("Mean squared error: " + str(mean_squared_error(y, y_pred)))
```

```
Adding x^2 term
```




```
Mean squared error: 0.6486438351011222
```

```
Adding sqrt(|x|) term
```




```
Mean squared error: 0.6492861409394708
```

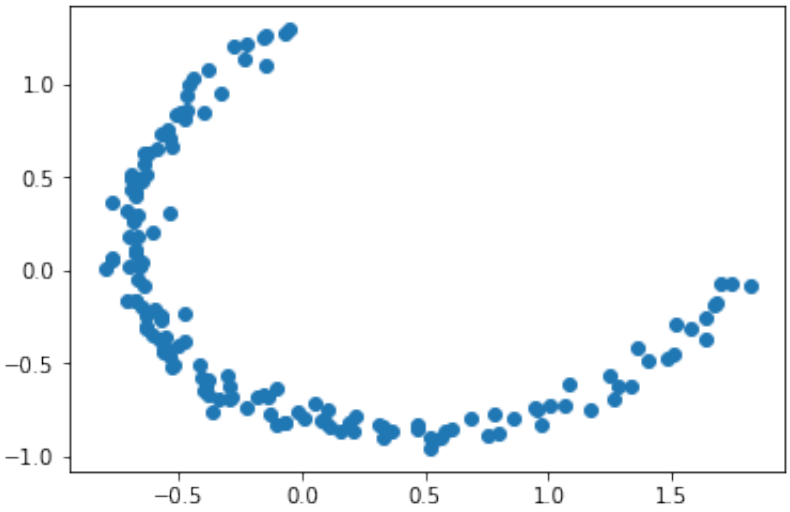
This is still not the best idea, please explain:

The data clearly doesn't follow neither a quadratic curve nor a sqrt(x) curve.

4. Plane Curves

As you've probably figured out, the above two methods are pretty crap at predicting orbits. What we really need to do is predict a curve in the plane. First, let's erase some of the data so what we're doing is actually a challenge. Just run the code in the next box:

In [78]:
```
# Create a mask where x < 0 or y < 0
def mask():
    global x
    global y

    mask = (x < 0) + (y < 0)
    x = x[mask]
    y = y[mask]

    # plot erased data
    plt.scatter(x, y)
    plt.show()

mask()
```



Now the most general form of a plane curve is

$$f(x, y) = 0$$

In order to simplify our lives a bit, let's restrict this to something of the form:

$$ax^2 + bxy + cy^2 + dx + ey + f = 0$$

You may recognize this as the general form of a conic! Let's take our data and try to predict the best possible coefficients here using least squares. This way, these coefficients should give the best possible approximation to the orbit. Print your predicted coefficients.

Hint 1: Think about the features you need. (6 total)

Hint 2: Use the normal equation instead of sklearn.

Hint 3: This is going to fail, why? a=b=c=d=e=f=0 works

In [94]:
```
### YOUR CODE HERE
feature_1 = np.square(x).reshape(-1, 1)
feature_2 = (x * y).reshape(-1, 1)
feature_3 = np.square(y).reshape(-1, 1)
feature_4 = x.reshape(-1, 1)
feature_5 = y.reshape(-1, 1)
feature_6 = np.ones(len(x)).reshape(-1, 1)
features = np.hstack((np.hstack((np.hstack((np.hstack((np.hstack((feat
ure_1, feature_2)), feature_3)), feature_4)), feature_5)), feature_6))

b = np.zeros(len(x))
x_hat = np.dot(np.matmul(np.linalg.inv(np.matmul(features.T, features)
), features.T), b)
print(x_hat)
```

```
[0. 0. 0. 0. 0. 0.]
```

5. Reformulation

The above should fail for a very trivial (pun intended) reason. The reason is that if we simply set all the coefficients to zero, we get a perfect solution! We can see this in the normal equations:

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

but $b = \vec{0}$ in our case, thus $x = \vec{0}$ trivially.

How do we get around this? One thing we can do is to not have $b = \vec{0}$. To do this, let us modify the general form of a plane curve a bit:

$$f(x, y) + 1 = 1$$

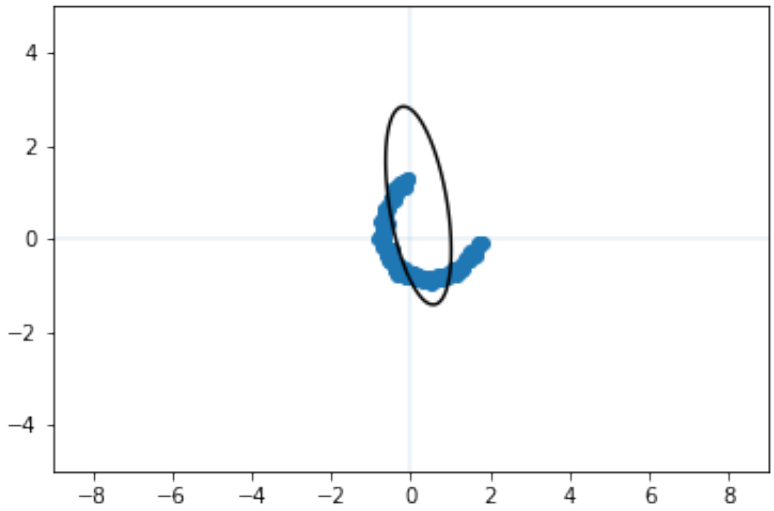
Now our restricted plane curve will be of the form

$$ax^2 + bxy + cy^2 + dx + ey + f + 1 = 1$$

Is this just an aesthetic change? or will this actually help? Code it up and find out! Plot your model using the handy dandy `plot_conic` function

In [95]:
```
# This function should help you plot your ellipses:

def plot_conic(coeff):
    '''
    params
    ------
    coeff : array[6] floats
        Array of 6 floats, corresponding to
        a, b, c, d, e, and f respectively
        in the equation above
    '''
    xv = np.linspace(-9, 9, 400)
    yv = np.linspace(-5, 5, 400)
    xv, yv = np.meshgrid(xv, yv)

    def axes():
        plt.axhline(0, alpha=.1)
        plt.axvline(0, alpha=.1)

    axes()
    plt.contour(xv, yv, xv*xv*coeff[0] + xv*yv*coeff[1] + yv*yv*coeff[
2] + xv*coeff[3] + yv*coeff[4] + coeff[5], [0], colors='k')
    plt.scatter(x,y)
    plt.show()
```

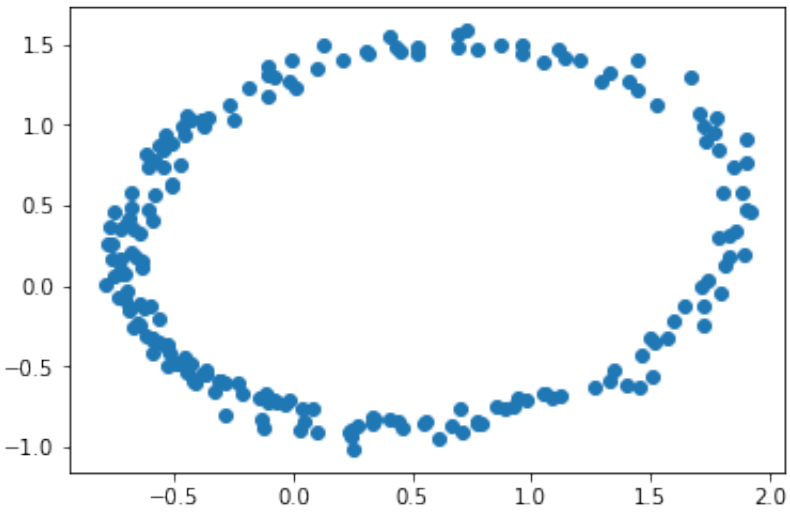
In [100]:
```
### YOUR CODE HERE
b = np.ones(len(x))
x_hat = np.dot(np.matmul(np.linalg.inv(np.matmul(features.T, features)
), features.T), b)
x_hat[5] -= 1
plot_conic(x_hat)
```

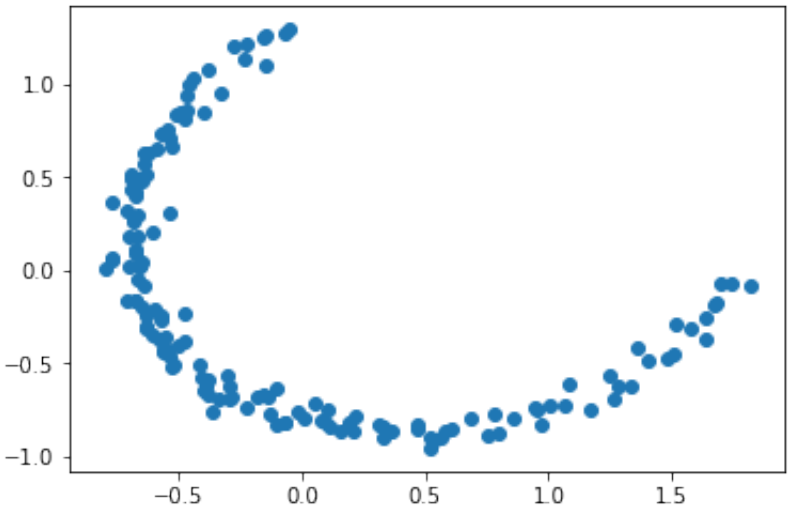


6. Ridge

So, reformulating the problem might have worked, but more than likely it didn't work too well. Here's some code to generate new data. Try running the above model multiple times on different data. More than likely most of them will look terrible.

In [102]:
```
# Regenerate data
gen_data(.5, 2, .5)
mask()
```





The problem here is that our method is too unstable. It turns out the Ridge Regression as a regularizer can reduce numerical instability and constrain under-constrained problems. (The math homework with the Ridge Derivation walks you through why this is the case)

The closed form solution for Ridge Regression is the following:

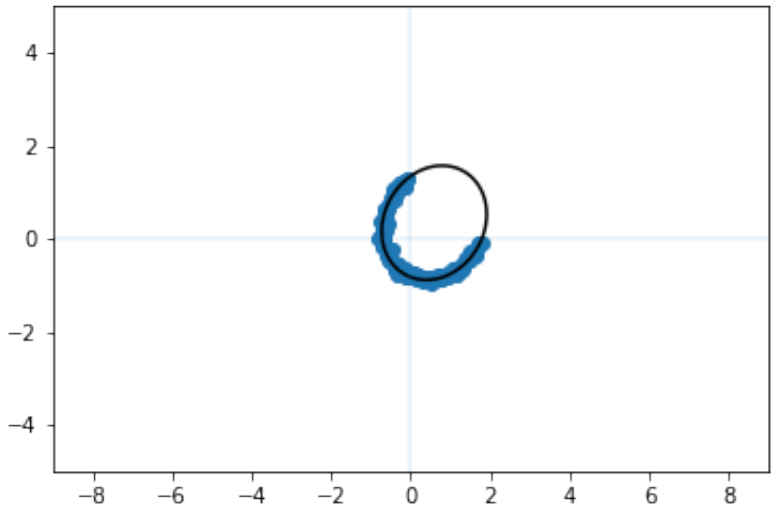
$$w^*_{\text{RIDGE}} = (X^T X + \lambda I)^{-1} X^T y$$

Rewrite the regression from above using ridge regression (try using $\lambda = 1$) and see how well it does. Plot out the model using `plot_conic` . Compare the results with the previous method.

Hint: Use the `regenerate data` block to try new data

Hint: There is really only one extra term between this question and the previous

In [103]:
```
### YOUR CODE HERE
b = np.ones(len(x))
ridge_lambda = 1
x_hat = np.dot(np.matmul(np.linalg.inv(np.matmul(features.T, features)
+ ridge_lambda * np.eye(features.shape[1])), features.T), b)
x_hat[5] -= 1
plot_conic(x_hat)
```



7. "Deriving" an Ellipse

LASSO regularization is a *sparse feature selector* in the sense that it zeros out "useless" features and keeps relevant features. It's a good way to reduce the number of features you have to use.

In this case we're going to pretend we don't know what form the equation of an ellipse takes. We can add random monomials to form a guess:

$$ax^2 + bxy + cy^2 + dx + ey + f + gx^3 + hy^3 + jx^2y + \cdots + 1 = 1$$

The idea here is that if we use LASSO regression on the above equation, the terms irrelevant to an ellipse will "zero out" and the quadratic and lower terms won't! Try this out, and print out the coefficients. No gurantees this will works 100% :), but you should find that all coefficients greater than quadratic zero out.

`Hint` : We want to keep the ridge regularization to maintain numerical stabilitiy. So we need a combined Ridge and LASSO regression. This model is called `ElasticNet` from sklearn. Use that model.

`Hint` : You might have to play around with the parameters a bit. I used these `l1_ratio=.23, alpha=.01` to produce some pretty good results

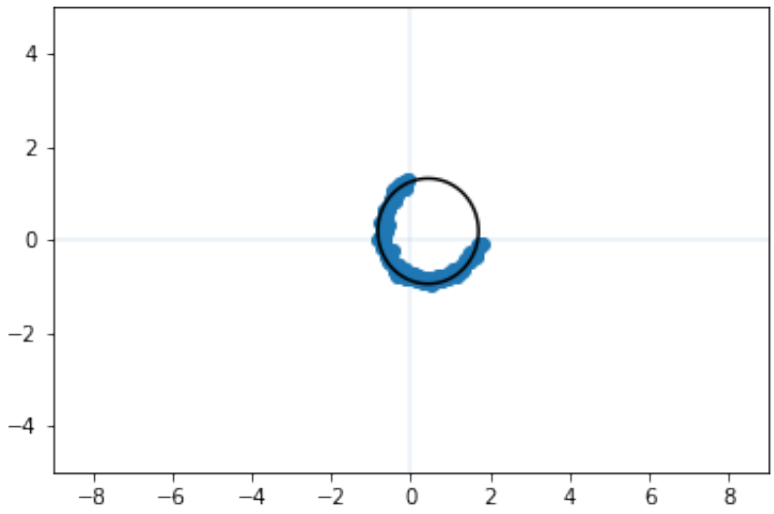
In [127]:
```
### YOUR CODE HERE
feature_7 = np.power(x, 3).reshape(-1, 1)
feature_8 = np.power(y, 3).reshape(-1, 1)
feature_9 = (np.power(x, 2) * y).reshape(-1, 1)
feature_10 = (x * np.power(y, 2)).reshape(-1, 1)
new_features = np.hstack((np.hstack((np.hstack((np.hstack((features, f
eature_7)), feature_8)), feature_9)), feature_10))

net = ElasticNet(l1_ratio=0.23, alpha=0.01, fit_intercept=False)
net.fit(new_features, b)

coeff = net.coef_[:]
print(coeff)
```

```
[ 0.05565683 -0.          0.06956352 -0.04951306 -0.02564773  0.9241
4928
 -0.         -0.         -0.          0.        ]
```

8. Evaluate this model!

Run this code block below. This code block assumes that you have an array called `coeff` which has 10 elements.

In [129]:
```
xv = np.linspace(-9, 9, 400)
yv = np.linspace(-5, 5, 400)
xv, yv = np.meshgrid(xv, yv)

def axes():
    plt.axhline(0, alpha=.1)
    plt.axvline(0, alpha=.1)

axes()
plt.contour(xv, yv, xv*xv*coeff[0] + xv*yv*coeff[1] + yv*yv*coeff[2] +
xv*coeff[3] + yv*coeff[4] + coeff[5] - 1 + coeff[6]*xv*xv*xv + coeff[7
]*yv*yv*yv + coeff[8]*xv*xv*yv + coeff[9]*xv*yv*yv , [0], colors='k')
plt.scatter(x,y)
plt.show()
```



As demonstrated above, ridge regression can help overcome numerical instability and generalization issues that ordinary least squares (OLS) can fall short to.