Homework 3

Qianlang Chen

Problem 1

```
import numpy

# read in the data for this problem
x = numpy.genfromtxt("data/x.csv")
y = numpy.genfromtxt("data/y.csv")
print(len(x), len(y))
```

```
100 100
```

Part (a)

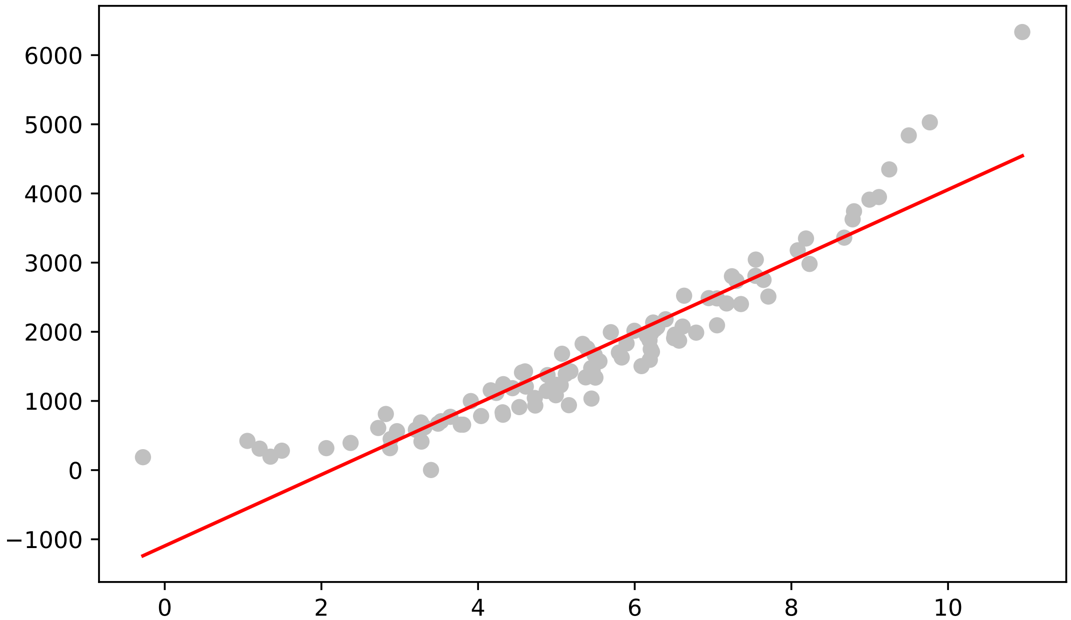
With the help of Python, let's find out the parameters of the regression line:

```
# line equation: y_hat = a*x + b
a, b = numpy.polyfit(x, y, 1)
print(a, b)
```

```
515.1222963490035 -1095.6026286820393
```

```
# plot the data and the regression line
from matplotlib import pyplot

pyplot.scatter(x, y, c="#C0C0C0")
line_x = numpy.linspace(min(x), max(x), 1729)
line_y = a * line_x + b
pyplot.plot(line_x, line_y, "r-", zorder=1729)
```



According to the output, the best-fit line for this data is approximately

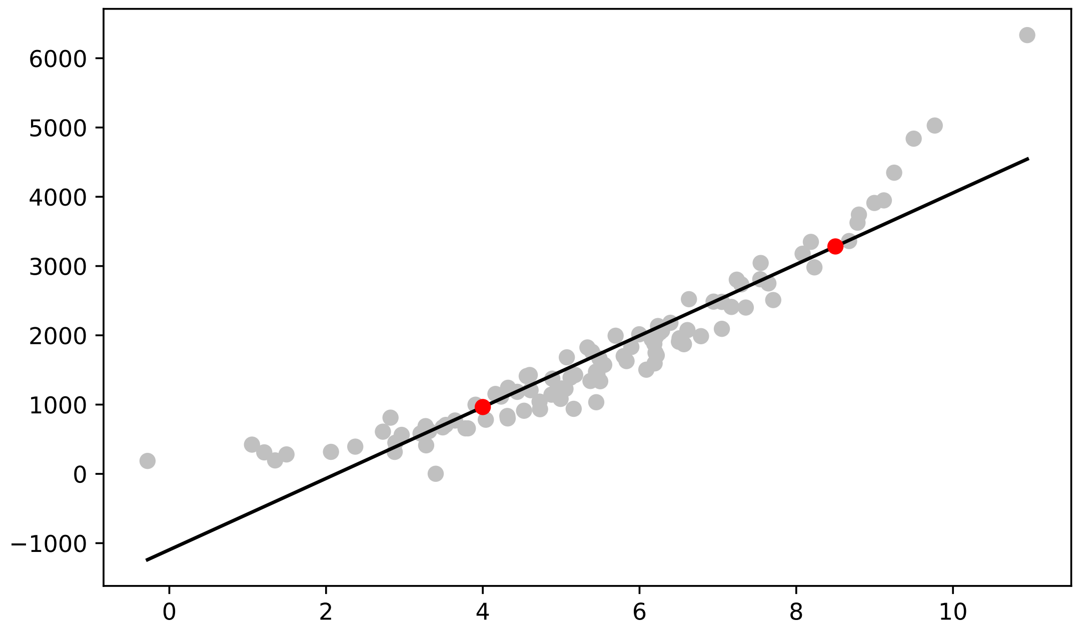
$$\hat{y} = M(x) = 515.1x - 1096$$

Using this model, we can then predict the values of y for $x = 4$ and $x = 8.5$:

```
# calculate and plot the predictions
pred_x = numpy.array([4, 8.5])
pred_y = a * pred_x + b
print(pred_x, pred_y)
```

```
[4.  8.5] [ 964.88655671 3282.93689028]
```

```
pyplot.scatter(x, y, c="#C0C0C0")
pyplot.plot(line_x, line_y, "k-")
pyplot.scatter(pred_x, pred_y, c="r", zorder=1729)
```



The output indicates that our predictions for inputs 4 and 8.5 are about 964.9 and 3283, respectively, and as seen from the plot, they do fit the data arguably nicely.

Part (b)

```
# split to get the training data and perform linear regression on it
train_len = int(.80 * len(x))
train_x = x[:train_len]
train_y = y[:train_len]
train_a, train_b = numpy.polyfit(train_x, train_y, 1)
print(train_len, train_a, train_b)
```

```
80 492.8393809405474 -991.1378015106925
```

The best-fit model for the training data is approximately

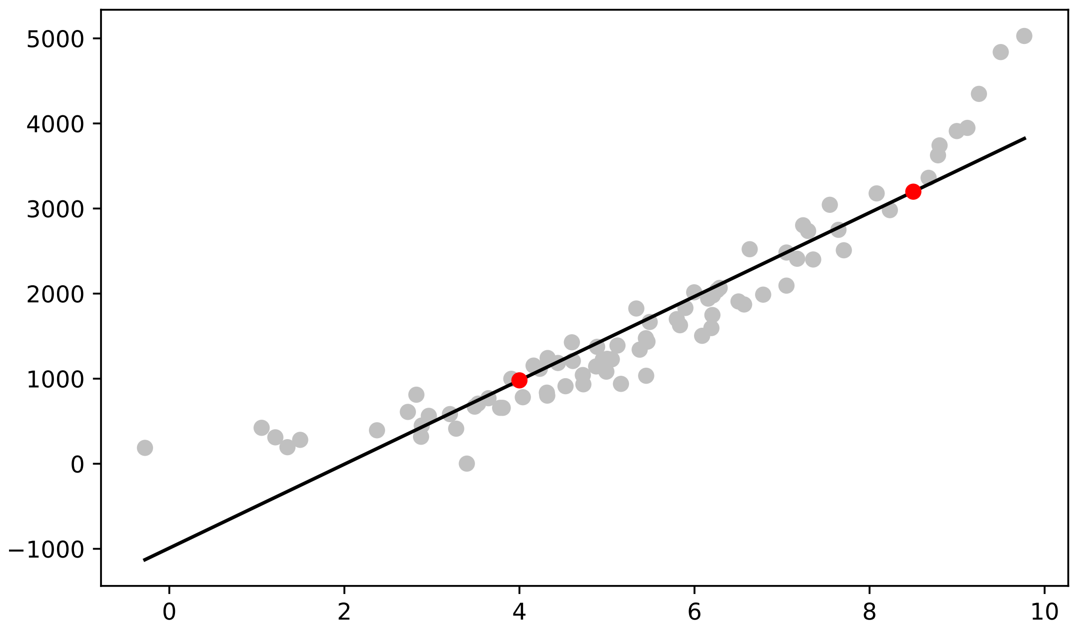
$$\hat{y} = M(x) = 492.8x - 991.1$$

```
# calculate and plot the predictions
pred_x = numpy.array([4, 8.5])
```

```
pred_y = train_a * pred_x + train_b
print(pred_x, pred_y)
```

```
[4.  8.5] [ 980.21972225 3197.99693648]
```

```
line_x = numpy.linspace(min(train_x), max(train_x), 1729)
line_y = train_a * line_x + train_b
pyplot.scatter(train_x, train_y, c="#C0C0C0")
pyplot.plot(line_x, line_y, "k-")
pyplot.scatter(pred_x, pred_y, c="r", zorder=1729)
```



Our predictions this time were about 980.2 and 3198, which ain't too bad either.

Part (c)

```
# split to get the testing data
test_x = x[train_len:]
test_data_y = y[train_len:]

# run the tests and calculate the residuals
test_full_y = a * test_x + b
test_train_y = train_a * test_x + train_b
res_full = test_full_y - test_data_y
res_train = test_train_y - test_data_y
res_full_norm = numpy.linalg.norm(res_full)
res_train_norm = numpy.linalg.norm(res_train)
print(f"Residual of the testing data using model built from full"
      f"data:\n"
      f"{res_full}\n"
      f"L2-norm of this residual vector: {res_full_norm}\n\n"
      f"Residual of the testing data using model built from training"
      f" data:\n"
      f"{res_train}\n"
      f"L2-norm of this residual vector: {res_train_norm}")
```

```
Residual of the testing data using model built from fulldata:
[  215.89690803   -17.36418609   -20.12249288  -100.18110103
    -6.71460716  -350.63745562   234.75647375 -1789.87014063
   144.22127685   -78.39281636  -227.1982354     17.23207118
   399.06602318    -4.66010069  -165.00458314   300.50641249
  -156.03643754  -157.72442192   397.35006545   189.12760159]
L2-norm of this residual vector: 2009.387840861227

Residual of the testing data using model built from training data:
[  182.45148582   -51.85145805   -83.69546569   -68.59127319
    23.87084202  -292.13940793   191.89664509 -1929.35492764
   133.28875434   -94.12670145  -305.15792262   -20.78470372
   381.01044539   -54.94514931  -173.56777438   259.8903937
  -178.47588917  -154.88998906   363.21463011   169.93289968]
```

```
L2-norm of this residual vector: 2115.5423038301
```

```
# calculate the residuals for the models built on the full data and the
#   training data
train_full_y = a * train_x + b
train_train_y = train_a * train_x + train_b
res_train_full = train_full_y - train_y
res_train_train = train_train_y - train_y
res_train_full_norm = numpy.linalg.norm(res_train_full)
res_train_train_norm = numpy.linalg.norm(res_train_train)
print(f"L2-norm of the residuals of the model built from full data:\n"
      f"{res_train_full_norm}\n\n"
      f"L2-norm of the residuals of the model built from training"
      f" data:\n"
      f"{res_train_train_norm}")
```

```
L2-norm of the residuals of the model built from full data:
3541.6022218903986

L2-norm of the residuals of the model built from training data:
3513.93821745281
```

Part (d)

```
# create tilde-x for a degree-3 polynomial regression
tilde_x = numpy.matrix([[x[i]**j for j in range(4)]
                       for i in range(train_len)])
print(f"First three rows of tilde-X:\n"
      f"{tilde_x[:3]}")
```

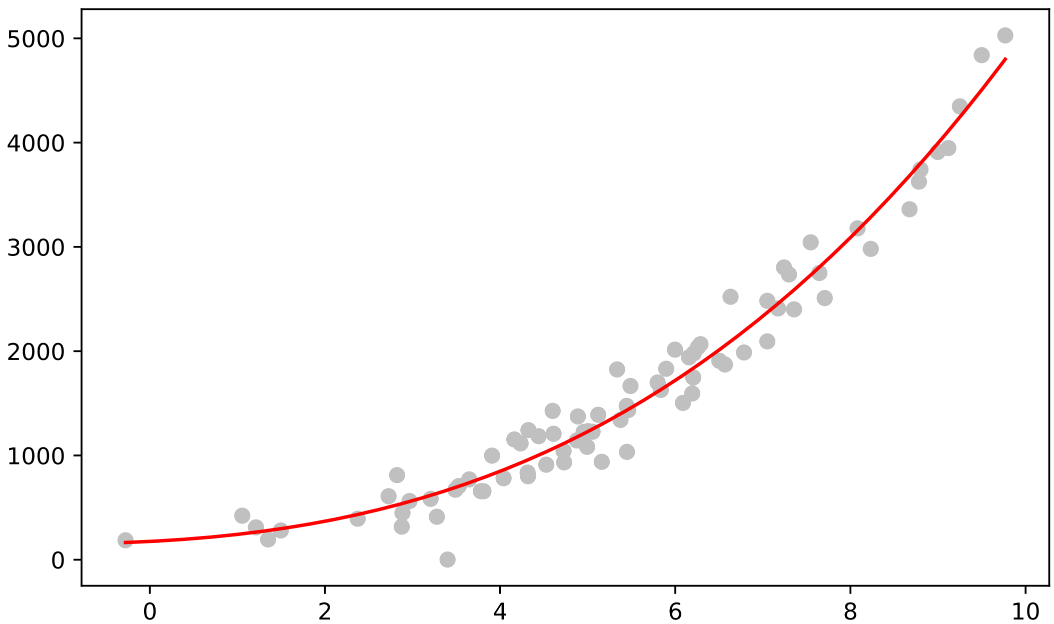
```
First three rows of tilde-X:
[[  1.           3.64641412  13.29633592  48.483947  ]
 [  1.           9.76756879  95.40540001 931.8788072 ]
 [  1.           2.72543008   7.42796911  20.24441042]]
```

```
# perform the polynomial regression and calculate the best-bit curve
coeffs = (tilde_x.T * tilde_x).I * tilde_x.T * numpy.matrix([train_y]).T
poly = numpy.poly1d(numpy.flip(numpy.squeeze(numpy.asarray(coeffs))))
```

```
print(poly)
```

```
       3         2
2.209 x + 22.49 x + 42.99 x + 175.4
```

```
# plot the best-fit line
line_x = numpy.linspace(min(train_x), max(train_x), 1729)
line_y = poly(line_x)
pyplot.scatter(train_x, train_y, c="#C0C0C0")
pyplot.plot(line_x, line_y, "r-")
```



The best-fit degree-3 polynomial model for the training data is

$$\hat{y} = M_3(x) = 2.209x^3 + 22.49x^2 + 42.99x + 175.4$$

```
# calculate the residuals
res_test = poly(test_x) - test_data_y
res_train = poly(train_x) - train_y
res_test_norm = numpy.linalg.norm(res_test)
res_train_norm = numpy.linalg.norm(res_train)
```

```python
print(f"L2-norm of the residual of the testing data: {res_test_norm}\n"
      f"L2-norm of the residual of the training data: {res_train_norm}")
```

```
L2-norm of the residual of the testing data: 942.2002113309505
L2-norm of the residual of the training data: 1831.546668230139
```

Problem 2

Part (a)

Yes. According to the Invertible Matrix Theorem, since the columns of X are linearly independent, they span $\mathbb{R}^n$.

Part (b)

Yes. According to the Invertible Matrix Theorem, since the columns of X are linearly independent, X is invertible.

Part (c)

The value of $||X\hat{\alpha} - y||_2^2 = 0$. Since X is invertible, Reginald definitely took the inverse of X to find the solution $\alpha = \hat{\alpha}$ to the equation $X\alpha = y$ because otherwise $\hat{\alpha}$ wouldn't have been the the model that minimizes the SSE to its smallest. Therefore, the SSE is zero since $X\hat{\alpha}$ matches y perfectly.

Part (d)

No, not necessarily. The model might be over-fitting the training data, meaning that it might be forced into weird shapes that captures no sensible patterns for the data, even if it fits the training data nicely.

Problem 3

```python
# define the functions we're interested in, along with their gradients
def f1(alpha):
    x, y = alpha[0], alpha[1]
    return (x - y)**2 + x*y


def grad_f1(alpha):
    x, y = alpha[0], alpha[1]
    return numpy.array([2*x - y, -x + 2*y])


def f2(alpha):
    x, y = alpha[0], alpha[1]
    return (1 - (y - 4))**2 + 35 * ((x + 6) - (y - 4)**2)**2


def grad_f2(alpha):
    x, y = alpha[0], alpha[1]
    return numpy.array([
        70*x - 70*(y - 4)**2 + 420,
        2*y - 140*(y - 4)*(x - (y - 4)**2 + 6) - 10,
    ])


# define the gradient descent algorithm
def grad_descent(num_iter, alpha, gamma, grad):
    values = numpy.zeros([num_iter, len(alpha)])
    values[0,:] = alpha
    for i in range(1, num_iter):
        alpha = alpha - gamma * grad(alpha)
        values[i,:] = alpha

    return values
```

```python
# define a helper function that creates the contour lines
def create_contour_lines(values, f, offset_x, offset_y):
    min_x = min(values[:,0]) - offset_x
    max_x = max(values[:,0]) + offset_x
    min_y = min(values[:,1]) - offset_y
    max_y = max(values[:,1]) + offset_y
    x, y = numpy.meshgrid(numpy.linspace(min_x, max_x, 1729),
                          numpy.linspace(min_y, max_y, 1729))
    z = f(numpy.array([x, y]))
    pyplot.contour(x, y, z)
```

Part (a)

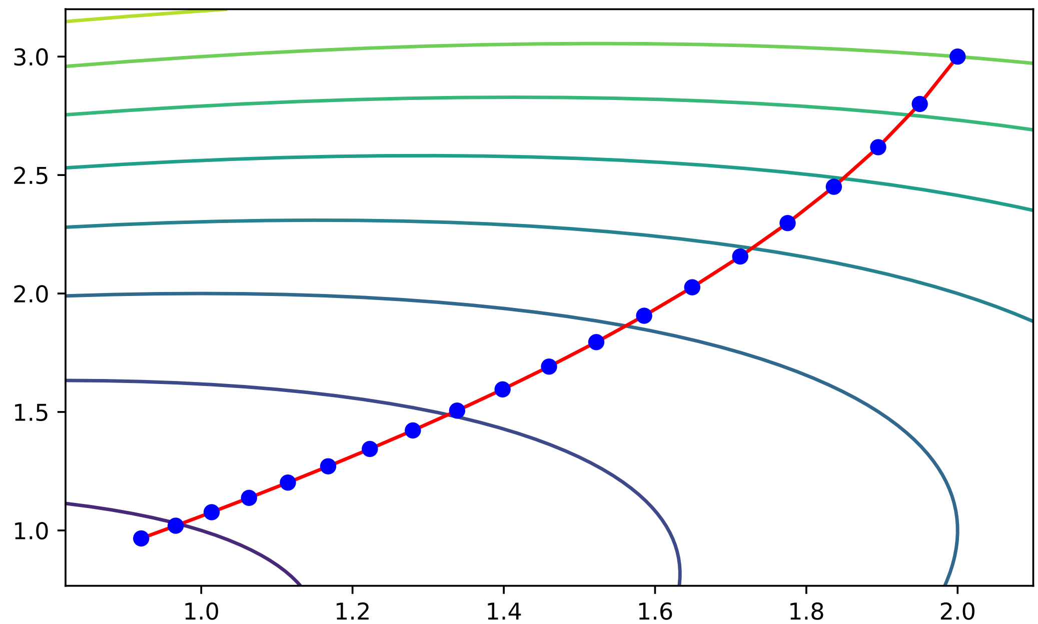
```python
# define the starting location and the learning rate
alpha = numpy.array([2, 3])
gamma = .05

# run the gradient descent algorithm
values = grad_descent(20, alpha, gamma, grad_f1)
print(values)
```

```
[[2.         3.        ]
 [1.95       2.8       ]
 [1.895      2.6175    ]
 [1.836375   2.4505    ]
 [1.7752625  2.29726875]
 [1.71259969 2.156305  ]
 [1.64915497 2.02630448]
 [1.5855547  1.90613178]
 [1.52230582 1.79479634]
 [1.45981505 1.691432  ]
 [1.39840515 1.59527955]
 [1.33832861 1.50567185]
 [1.27977934 1.4220211 ]
 [1.22290246 1.34380795]
```

```
[1.16780261 1.27057228]
[1.11455097 1.20190518]
[1.06319113 1.13744221]
[1.01374413 1.07685755]
[0.96621259 1.019859  ]
[0.92058428 0.96618373]]
```

```
# plot the results
create_contour_lines(values, f1, .1, .2)
pyplot.plot(values[:,0], values[:,1], "r-")
pyplot.plot(values[:,0], values[:,1], "bo")
```



Part (b)

```
# define the starting location and the learning rate
alpha = numpy.array([0, 2])
gamma = .0015

# run the gradient descent algorithm
values = grad_descent(100, alpha, gamma, grad_f2)
```

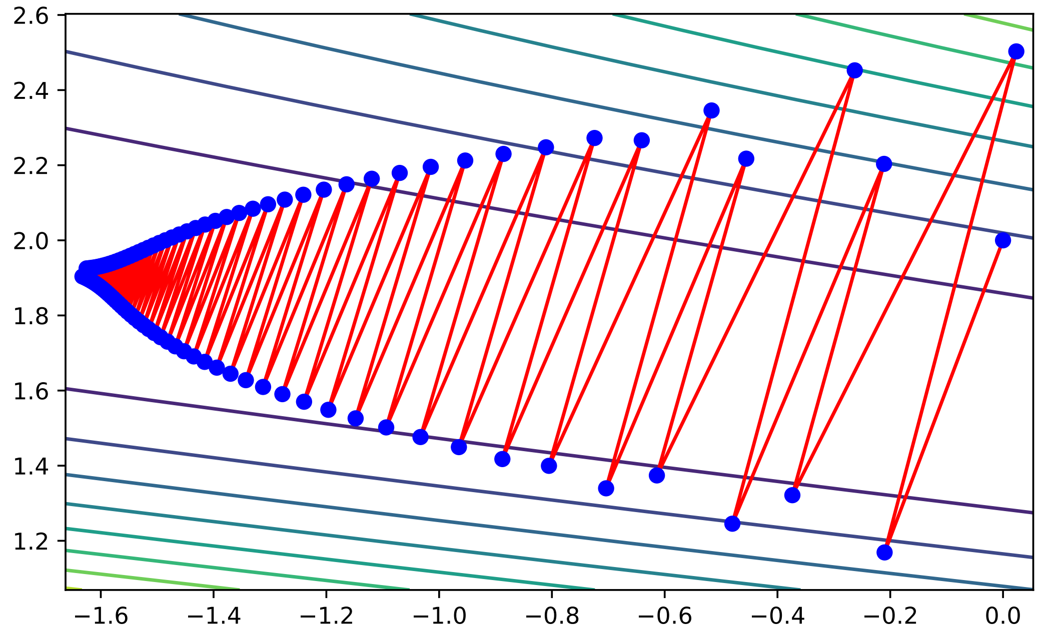
```python
print(values)
```

```
[[ 0.          2.        ]
 [-0.21        1.169     ]
 [ 0.02357891  2.50301676]
 [-0.3735962   1.32137875]
 [-0.21099236  2.20352283]
 [-0.47996849  1.24549331]
 [-0.26290455  2.45256496]
 [-0.61387128  1.37401085]
 [-0.4553538   2.21741919]
 [-0.70389424  1.33968008]
 [-0.51686863  2.34575698]
 [-0.80526282  1.39957153]
 [-0.64067626  2.26636397]
 [-0.8878284   1.41762119]
 [-0.72439498  2.27246231]
 [-0.96497293  1.44942998]
 [-0.81058299  2.24764638]
 [-1.03304374  1.47624364]
 [-0.8857928   2.23006589]
 [-1.09385455  1.50186453]
 [-0.95372833  2.21246747]
 [-1.14808324  1.52599863]
 [-1.01486281  2.1955957 ]
 [-1.19643535  1.54874836]
 [-1.06990301  2.17942735]
 [-1.23954229  1.57020779]
 [-1.11948188  2.16394086]
 [-1.2779694   1.59046412]
 [-1.16416698  2.14911475]
 [-1.31222295  1.6095982 ]
 [-1.20446736  2.13492771]
 [-1.34275635  1.627685  ]
 [-1.2408397   2.12135876]
 [-1.36997578  1.64479397]
```

```
[-1.2736938   2.10838738]
[-1.39424513  1.66098938]
[-1.30339747  2.09599369]
[-1.41589053  1.67633071]
[-1.33028093  2.08415851]
[-1.43520433  1.69087284]
[-1.35464071  2.07286351]
[-1.45244865  1.7046664 ]
[-1.37674313  2.06209127]
[-1.46785862  1.71775797]
[-1.39682745  2.05182533]
[-1.48164519  1.73019031]
[-1.41510867  2.04205025]
[-1.4939977   1.74200258]
[-1.43177994  2.03275164]
[-1.50508611  1.75323052]
[-1.44701489  2.0239161 ]
[-1.51506303  1.76390668]
[-1.46096951  2.01553127]
[-1.52406552  1.77406058]
[-1.47378397  2.00758573]
[-1.53221662  1.78371889]
[-1.48558417  2.00006897]
[-1.5396268   1.7929057 ]
[-1.49648314  1.99297126]
[-1.54639515  1.80164266]
[-1.50658228  1.98628354]
[-1.55261048  1.80994923]
[-1.51597253  1.97999732]
[-1.55835227  1.81784292]
[-1.52473529  1.97410449]
[-1.56369156  1.82533951]
[-1.53294338  1.96859719]
[-1.5686916   1.83245333]
[-1.54066183  1.96346763]
[-1.57340861  1.83919748]
```

```
 [-1.54794861  1.95870793]
 [-1.57789231  1.84558414]
 [-1.55485531  1.95430995]
 [-1.58218648  1.8516248 ]
 [-1.56142772  1.95026514]
 [-1.58632945  1.85733048]
 [-1.56770642  1.94656446]
 [-1.59035451  1.862712  ]
 [-1.57372729  1.94319819]
 [-1.59429039  1.86778016]
 [-1.57952194  1.94015595]
 [-1.5981616   1.8725459 ]
 [-1.58511824  1.93742659]
 [-1.60198887  1.87702047]
 [-1.59054062  1.93499821]
 [-1.60578945  1.88121548]
 [-1.59581054  1.93285818]
 [-1.60957752  1.88514298]
 [-1.60094676  1.93099321]
 [-1.61336449  1.88881547]
 [-1.60596571  1.92938942]
 [-1.61715935  1.89224591]
 [-1.61088175  1.92803245]
 [-1.62096897  1.89544759]
 [-1.61570743  1.92690758]
 [-1.62479837  1.89843414]
 [-1.62045374  1.9259999 ]
 [-1.62865107  1.90121936]
 [-1.62513029  1.92529439]
 [-1.63252926  1.90381715]]
```

```
# plot the results
create_contour_lines(values, f2, .03, .1)
pyplot.plot(values[:,0], values[:,1], "r-")
pyplot.plot(values[:,0], values[:,1], "bo")
```




```
# store the final result for comparison with part (d)
final_value_from_part_b = values[-1,:]
```

Part (c)

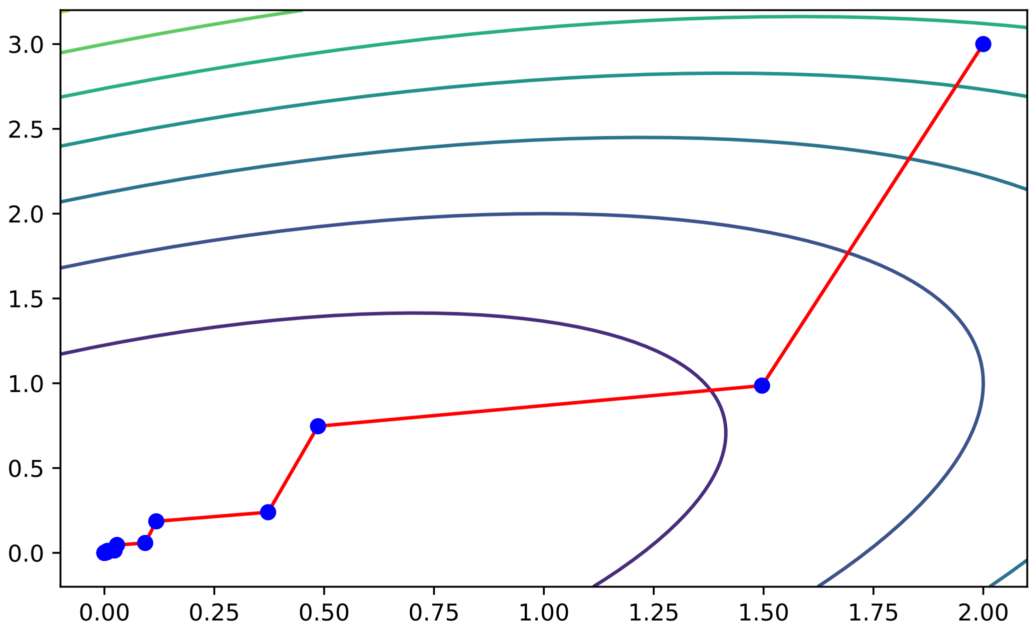
```
# this gamma value seems to work the best (brings alpha closest to the
#   true minimum after 20 iterations).
gamma = .503475293845

alpha = numpy.array([2, 3])
values = grad_descent(20, alpha, gamma, grad_f1)
final_value = values[-1,:]
print(f"Alpha value after 20 iterations: {final_value}\n"
      f"\"Cost\" of the above alpha value: {f1(final_value)}")
```

```
Alpha value after 20 iterations: [5.58807081e-06 2.76500461e-06]
"Cost" of the above alpha value: 2.3420744316033263e-11
```

```
create_contour_lines(values, f1, .1, .2)
pyplot.plot(values[:,0], values[:,1], "r-")
```

```
pyplot.plot(values[:,0], values[:,1], "bo")
```



Part (d)

To help reach the smallest value possible for f_2 within the 100-iteration limit, let's improve our gradient descent algorithm to make use of the "backtracking line search" technique, which automatically tunes the learning rate (γ) every time when the algorithm is about to make an overshooting step. As witnessed in the plot from *Part (b),* the original gradient descent algorithm wasn't very efficient: it was taking a lot of unnecessary overshooting steps when the steps are needed to be small, and overall it's only inching its way to the actual minimum, which was miles away. Hopefully our improved algorithm can find its descent more directly and smoothly with the assistance of backtracking line search.

Let's introduce this improved gradient descent algorithm. Note that this new algorithm is just like an ordinary gradient descent algorithm, except it ensures that no iteration will make the alpha value overshoot (i.e., go pass the minimum point) by shrinking the learning rate by some factor (β) when appropriate. This allows the descending process to take a big step when it's headed in the right direction and take a small step when its direction needs adjustments.

```python
# define a variant of the gradient descent algorithm which uses the
#   backtracking line search technique, with shrinking rate "beta" and
#   initial learning rate "gamma"
def grad_descent_w_bt_line_search(num_iter, alpha, beta, gamma, f,
                                  grad_f):
    values = numpy.zeros([num_iter, len(alpha)])
    values[0,:] = alpha
    gamma_0 = gamma
    for i in range(1, num_iter):
        gamma = gamma_0
        next_alpha = alpha - gamma * grad_f(alpha)
        # backtracking: shrink learning rate (gamma) if our next step
        #   is going to be too big
        while (f(next_alpha) > f(alpha) - gamma / 2
               * numpy.linalg.norm(grad_f(alpha))**2):
            gamma *= beta
            next_alpha = alpha - gamma * grad_f(alpha)

        alpha = next_alpha
        values[i,:] = alpha

    return values
```

Since the shrinking factor (β) plays a big role in the efficiency of the new algorithm, we'd like to know the best value possible for β. When we're doing gradient descent for real-world applications, we're often facing large amounts of data, where running the gradient descent algorithm can take a long time. In that case, we'd just try out several different β-values to estimate the best one to use. After trying out several different β-values, I found that the algorithm works the best (gives the lowest cost function value after 100 iterations) when $\beta = .92$.

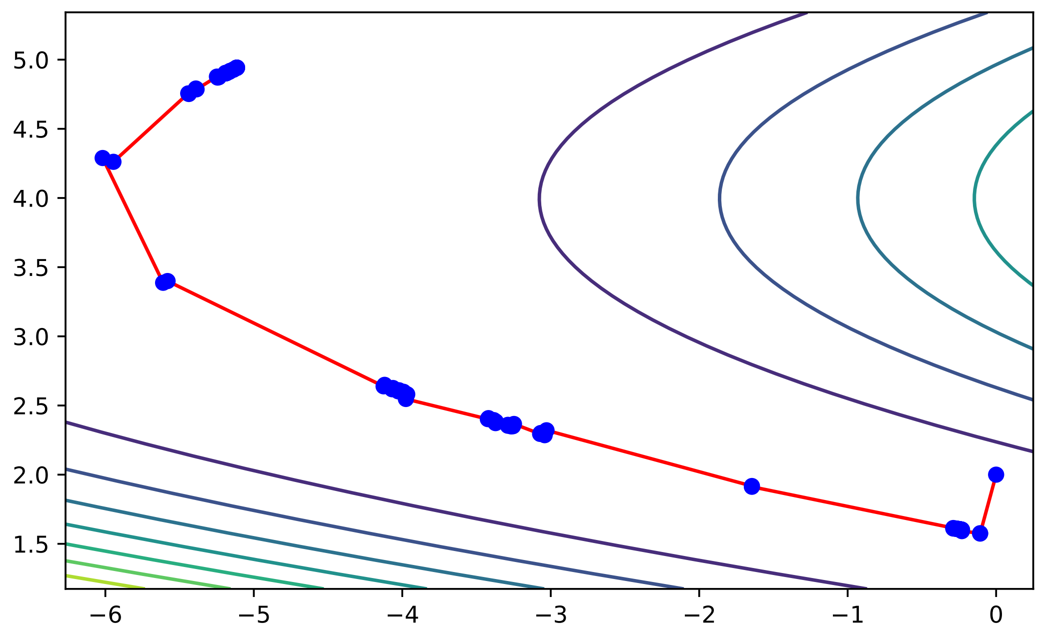
Now, knowing the best β-value to use, let's run the improved algorithm and see the results.

```python
# display and plot the results
alpha = numpy.array([0, 2])
beta = .92
gamma = 1
```

```
values = grad_descent_w_bt_line_search(100, alpha, beta, gamma, f2,
                                        grad_f2)
final_value = values[-1,:]
print(f"Alpha value after 100 iterations: {final_value}\n"
      f"\"Cost\" of the above alpha value: {f2(final_value)}")
```

```
Alpha value after 100 iterations: [-5.11295825  4.94229777]
"Cost" of the above alpha value: 0.003356857287640444
```

```
create_contour_lines(values, f2, .25, .4)
pyplot.plot(values[:,0], values[:,1], "r-")
pyplot.plot(values[:,0], values[:,1], "bo")
```



As seen from the plot, there's no more zigzagging going on, which signals a more efficient descending process.

```
# compare to the results from part (b)
print(f"Results from Part (b):\n"
      f"Alpha value after 100 iterations: {final_value_from_part_b}\n"
      f"\"Cost\" of the above alpha value:"
      f" {f2(final_value_from_part_b)}")
```

```
Results from Part (b):
Alpha value after 100 iterations: [-1.63252926  1.90381715]
"Cost" of the above alpha value: 9.610948838804072
```

So, the minimum point *Part (b)* was able to reach had a cost (by plugging into f_2) of 9.611, while the minimum we can now reach in *Part (d)* had a cost of only .003357, which is more than 2800 times smaller. That's quite an improvement!