Applied Modeling & Optimization

Exam 2

Group Members: Daniel Gural, Mourad Deihim, Jocelyn Ragukonis

I pledge my honor that I have abided by the Stevens Honor System.

Problem 1

Solve the following using steepest descent algorithm.

$$\text{minimize } f(x) = (x_1+5)^2 + (x_2+8)^2 + (x_3+7)^2 + 2x_1^2x_2^2 + 4x_1^2x_3^2$$

Start with x_0 and use stopping threshold $\in = 10^{-6}$.

$f(x)$ function

```
import numpy as np

np.random.seed(0)

x = np.random.uniform(low = -100.0, high = 100.0, size = (100, 3))

def f(x1, x2, x3):
  return (x1 + 5)**2 + (x2 + 8)**2 + (x3 + 7)**2 + 2 * x1**2 * x2**2 + 4 * x1**2 * x2**2

f = np.vectorize(f)
```

Steepest Descent Algorithm

A search for a stationary point can be viewed as an iterative procedure of generating a point $\boldsymbol{x}^{(k+1)}$ which takes steps of certain length t_k at direction $\Delta\boldsymbol{x}^{(k)}$ from the previous point $\boldsymbol{x}^{(k)}$.

The direction $\Delta\boldsymbol{x}^{(k)}$ decides which direction to search next, and the step size determines how far we go in that direction.

The update rule can be written as

$$\boldsymbol{x}^{(k+1)} = x^{(k)} + t_k\Delta\boldsymbol{x}^{(k)}$$

Given a particular point $\boldsymbol{x}$, the goal is to find the point that minimizes $f(\boldsymbol{x}+\boldsymbol{d})$

Using first-order Taylor expansion, $f(\boldsymbol{x}+\boldsymbol{d})$ can be approximated by

$$f(\boldsymbol{x}+\boldsymbol{d}) \approx f(\boldsymbol{x}) + \nabla f(\boldsymbol{x})^T \boldsymbol{d}$$

From the approximation, the direction $\boldsymbol{d}$ that minimizes $f(\boldsymbol{x}+\boldsymbol{d})$ is

$$\min_{\boldsymbol{d}:\|v\|=1} \nabla f(\boldsymbol{x}^T d)$$

Using the ℓ_2 norm, the steepest direction is

$$d^* = \frac{\nabla f(\boldsymbol{x})}{\|\nabla f(\boldsymbol{x})\|}$$

To minimize the function, $\boldsymbol{x}$ is updated at every iteration following

$$\boldsymbol{x}^{(k+1)} = x^{(k)} - t_k \nabla f(\boldsymbol{x}^{(k)})$$

Logic for Steepest Descent Algorithm

```
Guess $x^{(0)}$, set $k \leftarrow 0$
$\textbf{while}$ $\|\nabla f(\boldsymbol{x}^{(k)}) \geq \epsilon$ $\textbf{do}$
    $i \leftarrow \text{argmax}_{i\in[n]} \left| \frac{\delta f(\boldsymbol{x})}{\delta x_i} \right|$
    $\boldsymbol{x}^{(\boldsymbol{k}+\mathbf{1})} = \boldsymbol{x}^{(k)}$
```

Gradient of $f(x)$

$$\nabla f(x) = \begin{bmatrix} \frac{\delta f}{\delta x_1} \\ \frac{\delta f}{\delta x_2} \\ \frac{\delta f}{\delta x_3} \end{bmatrix} = \begin{bmatrix} 2(2x_1x_2^2 + 4x_1x_3^2 + x_1 + 5) \\ 2(2x_1^2x_2 + x_2 + 8) \\ 2(4x_1^2 + x_3 + 7) \end{bmatrix}$$

Hessian of $f(x)$

$$H(x) = \begin{bmatrix} 4x_2^2 + 8x_3^2 & 8x_1x_2 & 16x_1x_3 \\ 8x_1x_2 & 4x_1^2 + 2 & 0 \\ 16x_1x_3 & 0 & 8x_1^2 + 2 \end{bmatrix}$$

Partial derivative and gradient functions

```python
def partial_x1(x1, x2, x3):
  return 2 * (2 * x1 * x2**2 + 4 * x1 * x3**2 + x1 + 5)

def partial_x2(x1, x2, x3):
  return 2 * (2 * x1**2 * x2 + x2 + 8)
```

```python
def partial_x3(x1, x2, x3):
  return 2 * (4 * x1**2 * x3 + x3 +7)

def compute_gradient(x1, x2, x3):
  dx1 = partial_x1(x1, x2, x3)
  dx2 = partial_x2(x1, x2, x3)
  dx3 = partial_x3(x1, x2, x3)

  return np.array([dx1, dx2, dx3])
```

Hessian function

```python
def compute_Hessian(x1, x2, x3):
  H = np.array([[4 * x2**2 + 8 * np.square(x3) + 2, 8 * x1 * x2, 16 * x1 * x3],
          [8 * x1 * x2, 4 * x1**2 + 2, 0 ],
          [16 * x1 * x3, 0, 8 * x1**2 + 2]])
  return H
```

ℓ_2 norm function

```python
def l2_norm(x1, x2, x3):
  gradient = compute_gradient(x1, x2, x3)
  return np.linalg.norm(gradient)
```

Steepest descent function

```python
def run_steepest_descent(x, alpha, epsilon):
  """
    Inputs:
      x: the x values
      alpha: the step size
      epsilon: the stopping threshold
  """
  x1, x2, x3 = unpack(x)
  l2_norms = []
  iterations = 1

  gradient = compute_gradient(x1, x2, x3)
  norm = l2_norm(x1, x2, x3)
  l2_norms.append(norm)

  while norm >= epsilon: # While the l2 norm is greater than the stopping threshold
    x = steepest_descent_iteration(x, alpha)
    norm = l2_norm(x[0], x[1], x[2])
    l2_norms.append(norm)
    iterations += 1

  return x, iterations, l2_norms
```

```
def steepest_descent_iteration(x, alpha):
  x1, x2, x3 = unpack(x)
  gradient = compute_gradient(x1, x2, x3)
  new_x = x - alpha * gradient

  return new_x
```

Helper functions for steepest descent

```
def unpack(x):
  return x[0], x[1], x[2]
```

Results

```
x = np.array([1, 1, 1])
epsilon = 1e-6
alpha = 0.0001

x, iterations, l2_norms = run_steepest_descent(x, alpha, epsilon)
x
```

```
array([-0.015408  , -7.99620293, -6.99335859])
```

Verify that the final solution satisfies the second order necessary conditions for a minimum.

For a solution to satisfy the second order necessary conditions for a minimum, its Hessian must be positive semidefinite.

```
H = compute_Hessian(x[0], x[1], x[2])
np.linalg.cholesky(H)
```

```
array([[ 2.54757445e+01,  0.00000000e+00,  0.00000000e+00],
       [ 3.86895019e-02,  1.41402007e+00,  0.00000000e+00],
       [ 6.76745107e-02, -1.85166616e-03,  1.41326430e+00]])
```

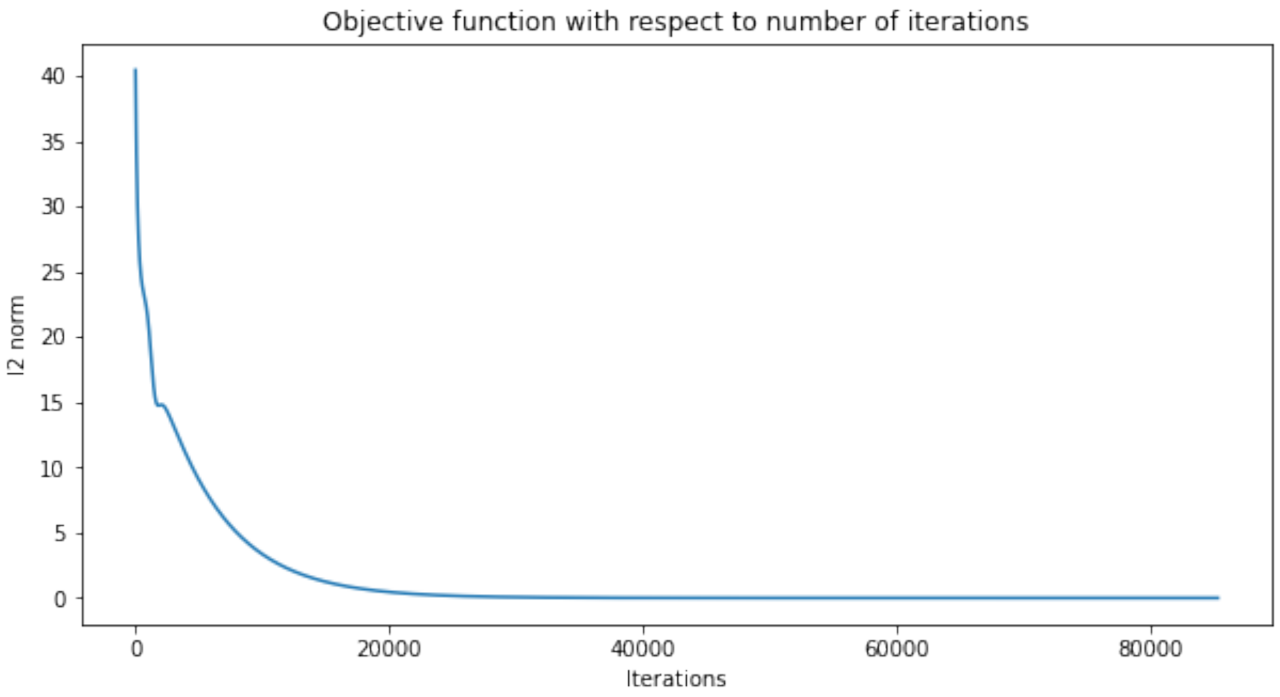
`np.linalg.cholesky(a)` returns the Cholesky decomposition of a square matrix `a`. `a` must be symmetric and positive-definite.

Since the the Cholesky decomposition of the Hessian of the solution was found, the solution is positive-definite.

Therefore, the final solution satisfies the second order necessary conditions for a minimum.

Plot the value of the objective function with respect to the number of iterations

```python
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

df = pd.DataFrame(data = l2_norms, columns = ['l2 norm'])
df['Iterations'] = np.arange(1, iterations + 1)

plt.figure(figsize = (10, 5))
sns.lineplot(data = df, x = 'Iterations', y = 'l2 norm')
plt.title('Objective function with respect to number of iterations')
```



Comment on the convergence speed of the algorithm

```python
import time

start = time.time()
x = np.array([1, 1, 1])
epsilon = 1e-6
alpha = 0.0001

x, iterations, l2_norms = run_steepest_descent(x, alpha, epsilon)
end = time.time()

print('The time to converge: ', end-start, ' seconds')
```

```
The time to converge:  2.6335854530334473  seconds
```

With a step size of 0.0001 and stopping threshold of 10^{-6} the steepest descent algorithm converged in about 2.6 seconds.

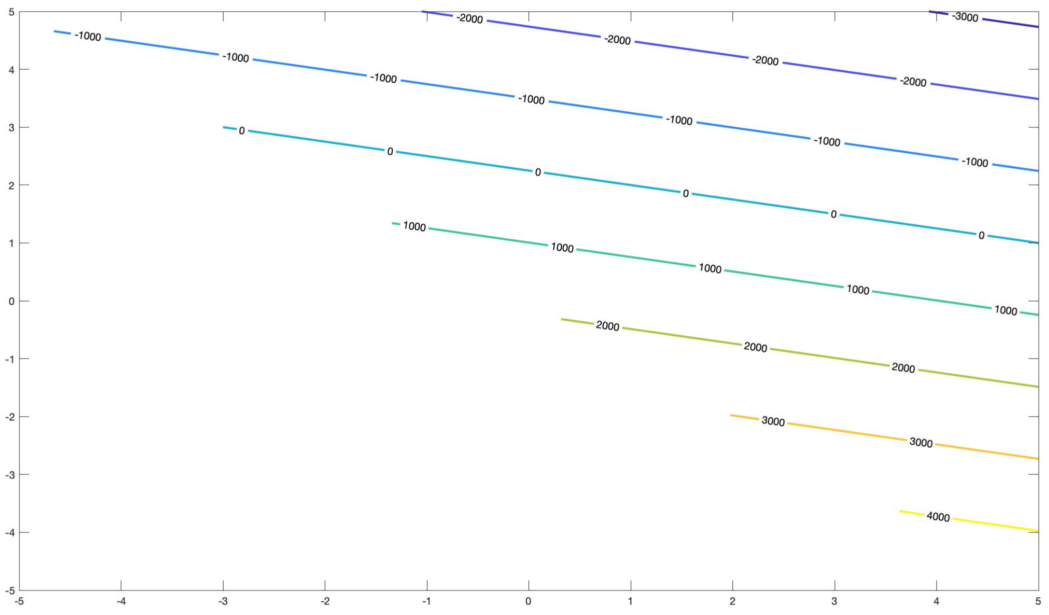
Problem 2

Consider the problem

$$\min f(x) \;\text{ such that } h(x) \geq 0$$
$$\text{where } \; f(x) = (x_1 - 1)^2 + 2(x_2 - 2)^2$$
$$\text{and } \; h(x) = \begin{bmatrix} 1 - x_1^2 - x_2^2 \\ x_1 + x_2 \end{bmatrix}$$

Plot the contour of $f(x)$ and the feasible set on one single figure, i.e. overlay the feasible set on the contour plot of $f(x)$.

```
cv = [.5;.5];
f = (cv(1)-1)^(2) + 2*(cv(2) - 2)^(2);
h = [1 - cv(1)^2-cv(2)^2, cv(1)+cv(2)];
h = h.';
x = -5:.05:5;
y = -5:.05:5;
[X,Y] = meshgrid(x,y);
F = (X-1)^(2) + 2*(Y - 2)^(2);


% Subdue the feasible set with an overlay
hf=fill([1 1 -1 -1]*5,[-1 1 1 -1]*5,'w','facealpha',0.8);

H1 =  X+Y;
H2 = 1 - X^2-Y^2;
feasible = (H1>=0 & H2 > 0);
F(~feasible) = NaN;
contour(X,Y,F,'linewidth',2,'ShowText','on');
```



Find a solution to the problem using the natural logarithmic barrier function, i.e., the barrier function is $-\log h_1(x) - \log h_2(x)$. Use initialization vector $[0.5, 0.5]^T$ and the initial penalty parameter equal to 1 and reduce it by $\frac{1}{2}$ in each iteration. Use a stopping threshold of 0.002.

Barrier function

```
clear
clc

format long
ipp = 1;
syms X Y;

% Barrier function
barrier = -ipp*(log(1 - X^2-Y^2) + log(X+Y));
minf = (X-1)^(2) + 2*(Y - 2)^(2);
f = minf + barrier;
```

Initialization

```
x(1) = .5;
y(1) = .5;
h1(1) = 1 - x(1)^2-y(1)^2;
h2(1) = x(1)+y(1);
values(1) =(x(1)-1)^(2) + 2*(y(1) - 2)^(2);
```

Gradient and Hessian

```
% Gradient and Hessian Computation:
df_dx = diff(f, X);
df_dy = diff(f, Y);

% Gradient
J = [subs(df_dx,[X,Y], [x(1),y(1)]) subs(df_dy, [X,Y], [x(1),y(1)])];
ddf_ddx = diff(df_dx,X);
ddf_ddy = diff(df_dy,Y);
ddf_dxdy = diff(df_dx,Y);
ddf_ddx_1 = subs(ddf_ddx, [X,Y], [x(1),y(1)]);
ddf_ddy_1 = subs(ddf_ddy, [X,Y], [x(1),y(1)]);
ddf_dxdy_1 = subs(ddf_dxdy, [X,Y], [x(1),y(1)]);

% Hessian
H = [ddf_ddx_1, ddf_dxdy_1; ddf_dxdy_1, ddf_ddy_1];
```

Solution

```
% Convergence Criteria
e = .002;

% Iteration Counter
i = 1;

% Search Direction
S = inv(H);

% Optimization Condition:
lastVal = 0;
while abs(values(i)-lastVal) > e && i<100
    I = [x(i),y(i)]';
    x(i+1) = I(1)-S(1,:)*J';
    y(i+1) = I(2)-S(2,:)*J';
    h1(i+1) = 1 - x(i+1)^2-y(i+1)^2;
    h2(i+1) = x(i+1)+y(i+1);
    values(i+1) =(x(i+1)-1)^(2) + 2*(y(i+1) - 2)^(2);
    lastVal = values(i);

    while h1(i+1) < 0 || h2(i+1)< 0 % Error, Boundary crossed, recalculate
        ipp = ipp*2;
        barrier = -ipp*(log(1 - X^2-Y^2) + log(X+Y));
        f = minf + barrier;
        df_dx = diff(f, X);
        df_dy = diff(f, Y);
        J = [subs(df_dx,[X,Y], [x(1),y(1)]) subs(df_dy, [X,Y], [x(1),y(1)])]; % Gradient
        ddf_ddx = diff(df_dx,X);
        ddf_ddy = diff(df_dy,Y);
        ddf_dxdy = diff(df_dx,Y);
        ddf_ddx_1 = subs(ddf_ddx, [X,Y], [x(1),y(1)]);
        ddf_ddy_1 = subs(ddf_ddy, [X,Y], [x(1),y(1)]);
        ddf_dxdy_1 = subs(ddf_dxdy, [X,Y], [x(1),y(1)]);
        H = [ddf_ddx_1, ddf_dxdy_1; ddf_dxdy_1, ddf_ddy_1]; % Hessian
        S = inv(H); % Search Direction
        x(i+1) = I(1)-S(1,:)*J';
        y(i+1) = I(2)-S(2,:)*J';
        h1(i+1) = 1 - x(i+1)^2-y(i+1)^2;
        h2(i+1) = x(i+1)+y(i+1);
        values(i+1) =(x(i+1)-1)^(2) + 2*(y(i+1) - 2)^(2);
        lastVal = values(i);
    end

    i = i+1;
    J = [subs(df_dx,[X,Y], [x(i),y(i)]) subs(df_dy, [X,Y], [x(i),y(i)])]; % Updated
Jacobian
    ddf_ddx_1 = subs(ddf_ddx, [X,Y], [x(i),y(i)]);
    ddf_ddy_1 = subs(ddf_ddy, [X,Y], [x(i),y(i)]);
    ddf_dxdy_1 = subs(ddf_dxdy, [X,Y], [x(i),y(i)]);
    H = [ddf_ddx_1, ddf_dxdy_1; ddf_dxdy_1, ddf_ddy_1]; % Updated Hessian
    S = inv(H); % New Search Direction

    ipp = ipp*.5;
    barrier = -ipp*(log(1 - X^2-Y^2) + log(X+Y));
    f = minf + barrier;
    df_dx = diff(f, X);
    df_dy = diff(f, Y);
end
```

Results

```
% Result Table:
Iter = 1:i;
X_coordinate = x';
Y_coordinate = y';
Iterations = Iter';
Values = values';
H1 = h1';
H2 = h2';
T = table(Iterations,X_coordinate,Y_coordinate,H1,H2,Values);
xEnd = X_coordinate(i);
yEnd = Y_coordinate(i);
value =(xEnd-1)^(2) + 2*(yEnd - 2)^(2);

figure('Color','w');
```

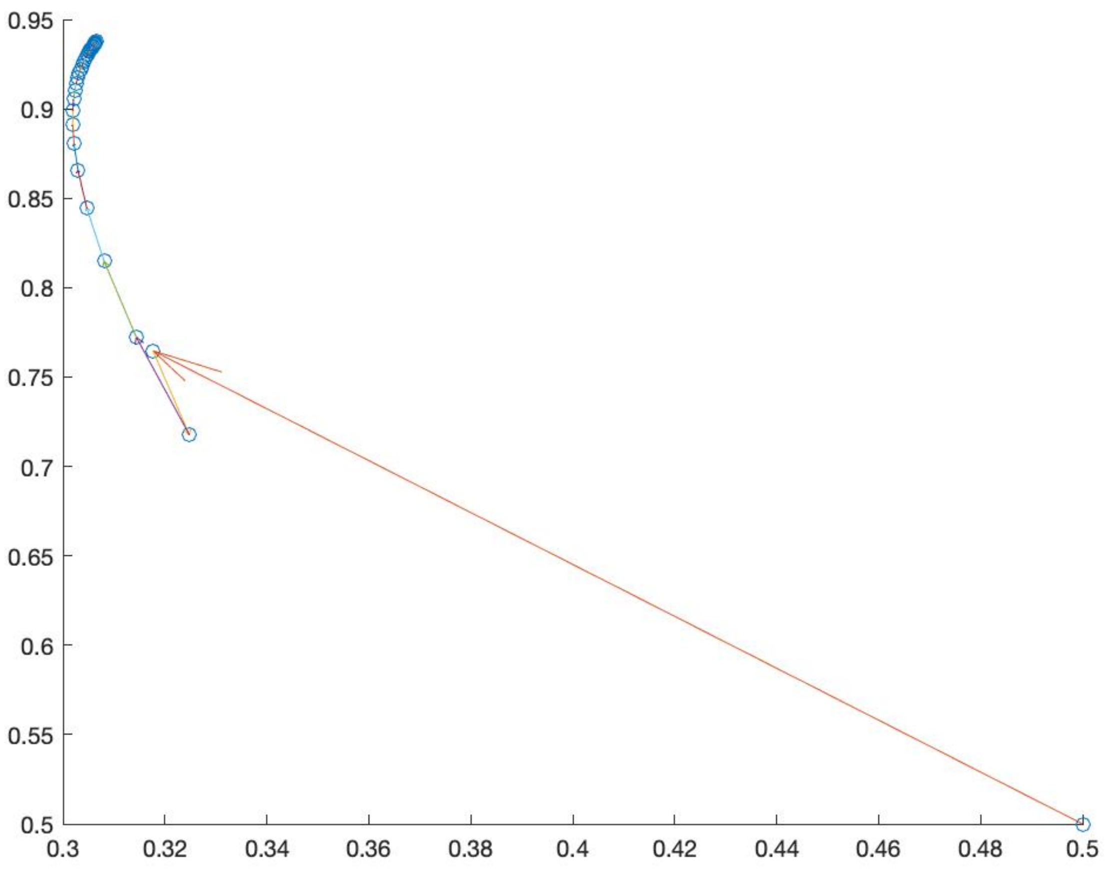
```
[X,Y] = meshgrid(x,y);
F = (X-1)^(2) + 2*(Y - 2)^(2);
```

The solution of $f(x)$, found after 33 iterations

x	$f(x)$	$h_1(x)$	$h_2(x)$
0.306556163064321	0.937684226217898	0.0267716107894245	1.24424038928222

In a 2-D figure, plot the trajectory (i.e, the values connected by lines with arrows) of the computed solution vector as the number of iteration progresses

```
% Plot scatter plot
scatter(X_coordinate,Y_coordinate);
hold on
for i=1:length(X_coordinate) - 1
    p1 = [X_coordinate(i) Y_coordinate(i)];
    p2 = [X_coordinate(i+1) Y_coordinate(i+1)];
    dp = p2 - p1;

    % Plot Arrows
    quiver(p1(1),p1(2),dp(1),dp(2),0);

    hold on
end

tab = T;
```



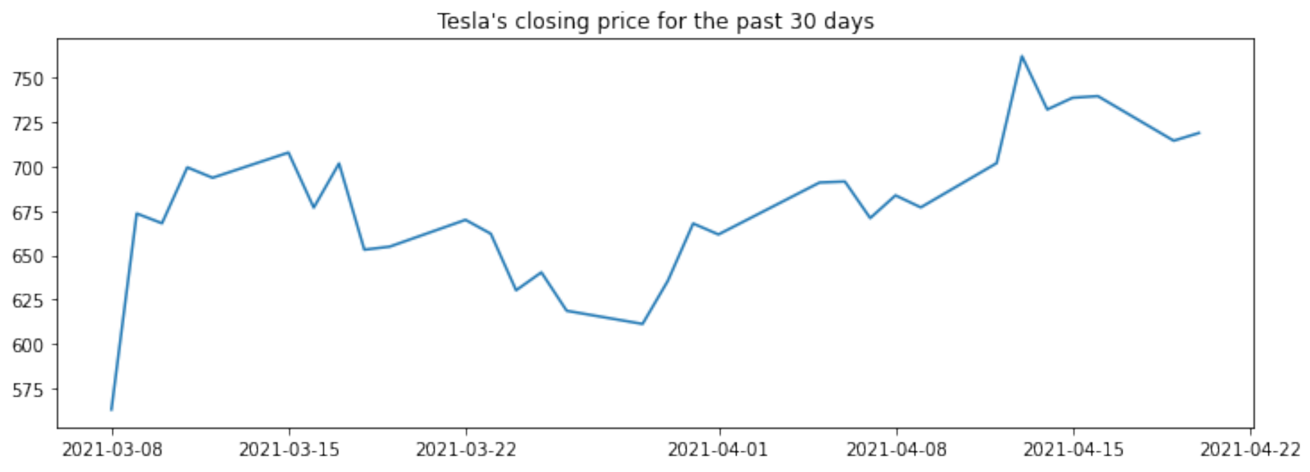
Problem 3

Collect the stock price for Tesla (NASDAQ: TSLA) for the past 30 days

```
import yfinance as yf

tickerSymbol = 'TSLA'

tickerData = yf.Ticker(tickerSymbol)

tickerDf = tickerData.history(period = '1d', start = '2021-03-08', end = '2021-04-21')

closing_price = tickerDf[['Close']]

tickerDf.head(5)
```

Plot the data (date vs stock price)

```
from matplotlib.dates import AutoDateFormatter, AutoDateLocator
import matplotlib.pyplot as plt

xtick_locator = AutoDateLocator()
xtick_formatter = AutoDateFormatter(xtick_locator)

plt.figure(figsize = (12, 4))
ax = plt.axes()
ax.xaxis.set_major_locator(xtick_locator)
ax.xaxis.set_major_formatter(xtick_formatter)
ax.plot(closing_price.index, closing_price['Close'])
plt.title('Tesla\'s closing price for the past 30 days')
```



Implement the stochastic gradient descent algorithm to fit a linear regression model for this data set.

Stochastic gradient descent algorithm

The objective function for stochastic gradient descent is

$$\begin{aligned} J(\theta, \theta_0) &= \frac{1}{n}\sum_{i=1}^{n} \text{Loss}(y^{(i)}(\theta \cdot x^{(i)} + \theta_0)) + \frac{\lambda}{2}\|\theta\|^2 \\ &= \frac{1}{n}\sum_{i=1}^{n} \left[\text{Loss}(y^{(i)}(\theta \cdot x^{(i)} + \theta_0)) + \frac{\lambda}{2}\|\theta\|^2\right] \end{aligned}$$

With stochastic gradient descent, choose $i \in \{1, \ldots, n\}$ at random and update θ such that

$$\theta \leftarrow \theta - \eta \nabla_\theta \left[\text{Loss}(y^{(i)}(\theta \cdot x^{(i)} + \theta)) + \frac{\lambda}{2}\|\theta\|^2\right]$$

Since we are trying to estimate a continuous-valued function, the cost function will be squared loss.

$$C = \frac{1}{2}(y - \hat{y})^2$$

Randomly initialize $\theta = 0$ (vector) and $\theta_0 = 0$ (scalar)

Using the previous day's price to predict the current day's price

$$y = \theta_0 + \theta_1 x_1$$

Using the past three days price to predict the current day's price

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3$$

Using the previous day's price

```
from sklearn.linear_model import SGDRegressor
from sklearn.preprocessing import StandardScaler

prices = closing_price['Close'].to_numpy().reshape(len(closing_price), )

x = prices[:-1].reshape(-1,1)
y = prices[1:].reshape(-1,1)

# Scale the x values
scaler = StandardScaler()
x_scaled = scaler.fit_transform(x)

# Stochastic Gradient Descent model using squared loss and l2-norm
sgd_model = SGDRegressor(loss = 'squared_loss', penalty = 'l2')
sgd_model.fit(x_scaled, y)

# Make predictions
predictions = sgd_model.predict(x_scaled)

predictions
```

```
array([612.45275096, 679.90819381, 676.5409004 , 695.7807526 ,
       692.19996893, 700.8682921 , 681.92123507, 697.12889863,
       667.45166555, 668.4948042 , 677.72433053, 672.94179841,
       653.48845594, 659.66180235, 646.43668677, 641.91035043,
       656.75202002, 676.4615955 , 672.69170874, 690.56513383,
       690.91284671, 678.31602698, 686.1425268 , 682.00664608,
       697.23259072, 734.04089786, 715.68553728, 719.72383203,
       720.29117852, 704.94929235])
```

Using the previous three days' prices

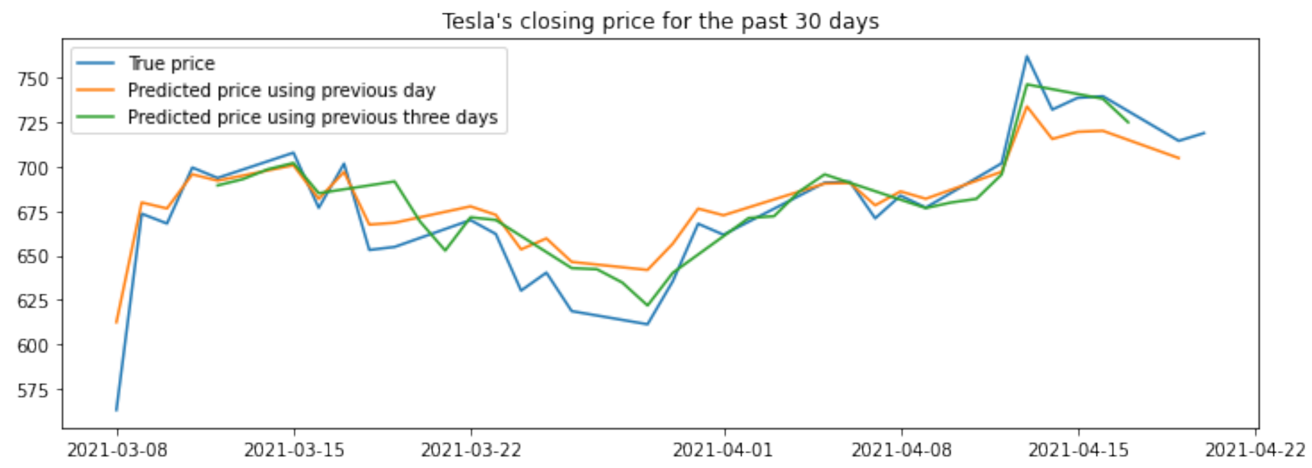
```
import numpy as np

# Function that takes a time series and transforms it
# Returns X and y values with X being the previous t days values

def create_dataset(dataset, t):
  dataX, dataY = [], []
  for i in range(len(dataset) - t - 1):
    a = dataset[i:(i + t), 0]
    dataX.append(a)
    dataY.append(dataset[i + t, 0])
  return np.array(dataX), np.array(dataY)

X, y = create_dataset(x, 3)

# Scale the x values
scaler_three_days = StandardScaler()
X_scaled = scaler_three_days.fit_transform(X)

# Stochastic Gradient Descent model using squared loss and l2-norm
sgd_model_three_days = SGDRegressor(loss = 'squared_loss', penalty = 'l2')
sgd_model_three_days.fit(X_scaled, y)
predictions_three_days = sgd_model_three_days.predict(X_scaled)
predictions_three_days
```

```
array([689.53538818, 693.05214984, 698.67993071, 702.15364551,
       685.13294874, 691.75898566, 669.25843948, 652.8315703 ,
       671.58162376, 670.05204414, 642.94160258, 642.29835045,
       634.80121309, 621.85214619, 640.32472164, 671.19398619,
       672.17366536, 686.10487673, 695.75363757, 676.75906206,
       679.89979288, 681.94991144, 695.67654994, 746.43482548,
       738.34256821, 725.12104652])
```

Plot the raw data and your linear regression model together for visual comparison

```
# Find the dates of the predictions
dates = closing_price.index.to_numpy()
dates_predictions_three_days = dates + np.timedelta64(4, 'D')

# Plot
xtick_locator = AutoDateLocator()
xtick_formatter = AutoDateFormatter(xtick_locator)
```

```
plt.figure(figsize = (12, 4))
ax = plt.axes()
ax.xaxis.set_major_locator(xtick_locator)
ax.xaxis.set_major_formatter(xtick_formatter)
ax.plot(closing_price.index, closing_price['Close'], label = 'True price')
ax.plot(dates[: predictions.shape[0]], predictions, label = 'Predicted price using
previous day')
ax.plot(dates_predictions_three_days[: predictions_three_days.shape[0]],
predictions_three_days, label = 'Predicted price using previous three days')
plt.legend()
plt.title('Tesla\'s closing price for the past 30 days')
```



What does your model predict for Tesla's stock price for the next three months into the future?

Future predictions using previous day

```
future_predictions = []

# Last available price
x = prices[-1].reshape(1, -1)

# Make predictions
for days in range(90):
  x = sgd_model.predict(x)
  future_predictions.append(x)
  x = scaler.transform(x.reshape(1, -1))

future_predictions = np.array(future_predictions)
print('On ', str(future_predictions_dates[-1])[:10], ' Tesla\'s price will be $',
round(future_predictions[-1][0], 2), ' using the previous day\'s price')
```

```
On  2021-07-19  Tesla's price will be $ 689.81  using the previous day's price
```

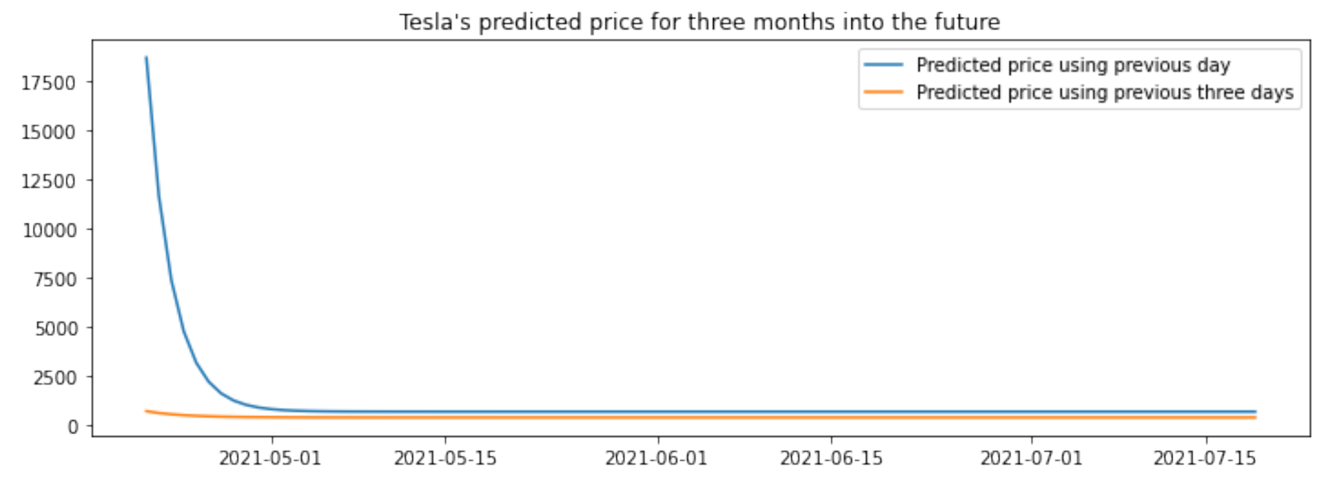
Future predictions using previous three days

```
# Last available prices
x = prices[len(prices) - 4: -1].reshape(1, -1)
x = scaler_three_days.transform(x)

future_predictions_three_days = []

# Make predictions
for days in range(90):
  y = sgd_model_three_days.predict(x)
  future_predictions_three_days.append(y)
  x = np.append(x[0][1:], y)
  x = scaler_three_days.transform(x.reshape(1,-1))

future_predictions_three_days = np.array(future_predictions_three_days)
print('On ', str(future_predictions_dates[-1])[:10], ' Tesla\'s price will be $',
round(future_predictions_three_days[-1][0], 2), ' using the previous three day\'s price')
```

```
On  2021-07-19  Tesla's price will be $ 393.97  using the previous three day's
price
```

Plot the predictions

```
# Find the dates for the predictions
future_predictions_dates = [dates[-1]]

for days in range(90):
  future_predictions_dates.append(future_predictions_dates[-1] + + np.timedelta64(1, 'D'))

future_predictions_dates = future_predictions_dates[1:]
future_predictions_dates = np.array(future_predictions_dates)

# Plot the predictions
xtick_locator = AutoDateLocator()
xtick_formatter = AutoDateFormatter(xtick_locator)

plt.figure(figsize = (12, 4))
ax = plt.axes()
ax.xaxis.set_major_locator(xtick_locator)
ax.xaxis.set_major_formatter(xtick_formatter)
ax.plot(future_predictions_dates, future_predictions, label = 'Predicted price using
previous day')
ax.plot(future_predictions_dates, future_predictions_three_days, label = 'Predicted price
using previous three days')
plt.legend()
plt.title('Tesla\'s predicted price for three months into the future')
```



The visual shows that the model is unable to predict future values. The technique used to predict the price is a feed-forward network, since we specified the window size of 1 or 3 time steps, i.e. the most recent 1 or 3 points were used as the feature vector. However, the number of time steps needed for better future predictions would need to be addressed.

Explain the pros and cons of gradient descent and stochastic gradient descent

Both gradient descent and stochastic gradient descent are both first order optimization algorithms to find the parameters that minimize a multivariable cost function F(x). With gradient descent, all samples of the data are used to calculate the gradient and update weights. On the other hand, stochastic gradient descent uses random samples to calculate the gradient and weights.

Gradient descent

Pros

- Better at finding local minima, as it doesn't skip over any samples and its steps continues in only one direction
- Works very well with convex functions with a smoother curve
- Will typically find more optimal minimum than sgd given the time to converge and tuned learning rate and iterations

Cons

- Requires more memory, since all of the data is being used to update the parameters. Updates are more computationally expensive as a result.
- More likely to get stuck at a local minimimum when trying to find global minimum.
- Slow to reach convergence.

Stochastic gradient descent

<u>Pros</u>

- Less likely to get stuck at local minima, because of the randomness of sample selections used to update parameters. This makes sgd better for data with lots of minima, and a more noisy curve.
- Less memory is required, since only one sample is being used at a time to calculate the gradient. Updates are not as computationally expensive, which allows the practical number of iterations to be greater.
- Typically converges much faster.

<u>Cons</u>

- Less likely to find local minima since each iteration jumps to a new random sample instead of moving in the same direction, which could've converged to a local minimum. Noisy steps can cause it to move away from the minimum.
- Less optimal minimum for smooth curves.