Assignment1_JacobJames_K

September 21, 2022

0.1 Question : 1

Linear Least Squares [5 points]

The simplest model to predict the spread of infectious diseases is the SIR model. This model is a set of ordinary differential equations that describe the evolution of the number of susceptible (S(t)), infected (I(t)) and recovered/removed (R(t)) populations in a closed system. The equations are

$dS\frac{}{dt=\frac{\beta IS}{\{}N\}\$,(1)}$

$\frac{dI}{dt} = \frac{\beta IS}{N}\gamma I$, (2)

$\frac{dR}{dt} = \gamma I$, (3)

where $N = S(t) + I(t) + R(t)$. The basic reproduction number $R_0 = \beta/\gamma$ is defined to quantify the new infections that one infected person causes and is considered as a magic number to identify if an infectious disease is under control. For example, if $R_0 > 1$, the disease has an exponential growth whereas if $R_0 < 1$, the disease is under control and the infectious population will eventually go to zero. At peak R_0 will cross 1. As with simple models, there exist analytical solutions to the SIR model. One form of the solution is given as

$S(t) = S(0)exp(\chi(t))$, (4)

$I(t) = NS(t)R(t)$, (5)

$R(t) = R(0) + \rho\chi(t)$, (6)

$\chi(t) = \frac{\beta}{N}\int_0^t I(t^*)dt^*$. (7)

For this assignment, we will consider a time unit of days, total population of India as 130 crores and the time horizon of interest as March 23, 2020 to Oct 15, 2020. Removed is a sum of recovered and deceased, i.e., the population that will not get infected again. Data in CSV form for state-wise-daily is available at https://data.covid19india.org/csv/latest/state_wise_daily.csv

1. Download the state wise daily data of infected, recovered and deceased from the covid19india website. The data gives daily new infections, recovery and deceased. Use Pandas and create time-series of all India $I(t), S(t)$ and $R(t)$. Plot these time- series. Hint: Apply yourself and see what $I(t)$ means and what the data provides.
2. Formulate the problem of estimating γ and β as a linear least squares problem.
3. Form the Jacobian matrix and calculate its rank and condition number.
4. Form the coefficient matrix and calculate its condition number. Find the relation between this condition number and condition number of the Jacobian.
5. Code the cholesky factorization approach to solve the linear least squares problem.

6. Apply your code and estimate β, γ and R_0.
7. Use scipy.optimize and estimate β, γ and R_0.
8. State your observations in the above two items and give reasons.
9. Estimate $R_0(t)$ as a function of time by utilizing data until t to estimate the different parameters. Plot $R_0(t)$.
10. Based on the above analyse the state of the pandemic in India. Has the peak passed as on Oct 2020?

[1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

POPULATION = 130 * (10**7)

df = pd.read_csv('./data/state_wise_daily.csv')
summed_data = df.sum(axis=1,numeric_only=True)    # Sum up along rows
```

Formula Used

- S(1) = Initial_population - Infected count from csv on day 1
- R(1) = Recovered count from csv on day 1
- I(1) = Infected count from csv on day 1 - R(1)

Further on - S(t) = S(t-1) - Infected count from csv on day t

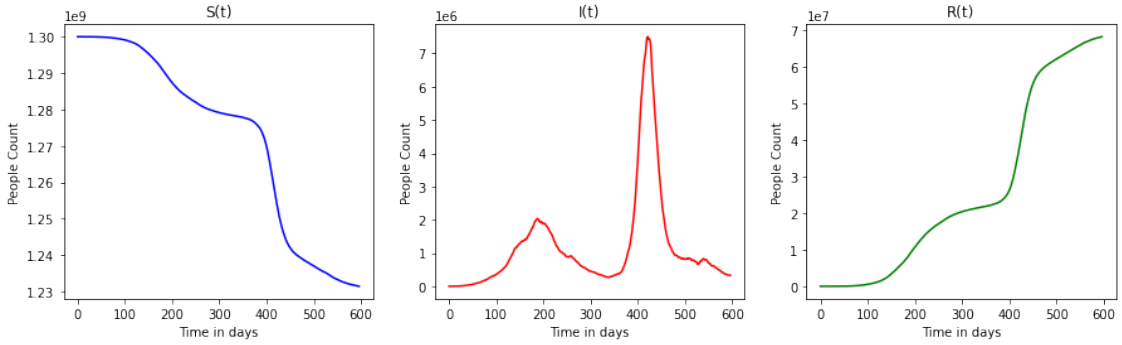
R(t) = R(t-1) + Recovered count from csv on day t

I(t) = I(t-1) + (Infected count from csv on day t - Recovered count on day t)

[2]:
```
def get_susceptible_infected_recovered(data_frame):


    infected_indices = [3*i for i in range(int(len(data_frame)/3))]
    recovered_indices = [ 3*i+1 for i in range(int(len(data_frame)/3)) ]   # Add␣
↪indices of Recovered
    recovered_indices =  recovered_indices +  [ 3*i+2 for i in␣
↪range(int(len(data_frame)/3)) ] # Add indices of Decesesed
    recovered_indices = sorted(recovered_indices)

    recovered_deceased = summed_data[recovered_indices]  # Get only rows related␣
↪to recovered and deceased
    recovered_dup = recovered_deceased.groupby(recovered_deceased.index//3).
↪transform('sum')
    rec_indices = [ 3*i + 1 for i in range(int(len(recovered_dup)//2))]


    recovered = recovered_dup[rec_indices]
    recovered = recovered.reset_index(drop = True)
    infected = summed_data[infected_indices]
```

```
    infected = infected.reset_index(drop=True)

    new_dataframe = pd.DataFrame({'infected' :  infected , 'recovered' :␣
↪recovered})
    time_series = new_dataframe.index.array



    time_series_df = pd.DataFrame()
    time_series_df.insert(0,"time",time_series)
    time_series_df["S(t)"] = 0
    time_series_df["I(t)"] = 0

    time_series_df.insert(3,"R(t)",new_dataframe['recovered'].cumsum()) # R(t)␣
↪increases as more people recover each day So taking cumulative sum

    time_series_df.loc[0,"S(t)"] = POPULATION- new_dataframe.loc[0,'infected'] #␣
↪Infected people will only get deducted from susceptible
    time_series_df.loc[0,"I(t)"] = new_dataframe.loc[0,'infected'] -␣
↪new_dataframe.loc[0,'recovered'] # Formuala derived from I(t) = N - S(t) -␣
↪R(t)

    # Continue for all timesteps
    for i in range(1,len(new_dataframe)):

        time_series_df.loc[i,"S(t)"] = time_series_df.loc[i-1,"S(t)"] -␣
↪new_dataframe.loc[i,'infected']
        time_series_df.loc[i,"I(t)"] = time_series_df.loc[i-1,"I(t)"] +␣
↪(new_dataframe.loc[i,'infected'] - new_dataframe.loc[i,'recovered'])
    return time_series_df


# To plot the time series graph for infected recovered and deceased.
def plot_time_series(data_frame):

    def set_axis_title(ax,title):
        ax.set_ylabel('People Count')
        ax.set_xlabel('Time in days')
        ax.set_title(title)

    figure,axes = plt.subplots(1,3)
    figure.set_figwidth(15)

    axes[0].plot(time_series_df['time'],time_series_df['S(t)'] ,color='b', label␣
↪= 'S(t)')
```

```
    axes[1].plot(time_series_df['time'],time_series_df['I(t)'] ,color='r', label␣
↪= 'I(t)')
    axes[2].plot(time_series_df['time'],time_series_df['R(t)'] ,color='g', label␣
↪= 'R(t)')
    set_axis_title(axes[0],'S(t)')
    set_axis_title(axes[1],'I(t)')
    set_axis_title(axes[2],'R(t)')
    plt.show()



time_series_df = get_susceptible_infected_recovered(summed_data)
plot_time_series(time_series_df)
```



Least Square Problem Formulation

- β and γ are the unknown parameters
- dS/dt , dI/dt and dR/dt at a particular time "t" can be estimated using the central difference method whose general form is $\frac{\partial f(t)}{dt} = \frac{f(t+1)-f(t-1)}{2}$
- Edge cases handled by taking forward difference only and vice versa

We have the data samples of I(t), S(t) and R(t) upto time T. we can find

$$\begin{bmatrix} \beta \\ \gamma \end{bmatrix}$$

at time T as the solution to

$$argmin_{\beta,\gamma} \; ||A \begin{bmatrix} \beta \\ \gamma \end{bmatrix} - b||_2^2$$

where A and b are

$$\begin{bmatrix} -\frac{I(1)*S(1)}{N} & 0 \\ \frac{I(1)*S(1)}{N} & -I(1) \\ 0 & I(1) \\ -\frac{I(2)*S(2)}{N} & 0 \\ \frac{I(2)*S(2)}{N} & -I(2) \\ 0 & I(2) \\ . & \\ . & \\ . & \\ -\frac{I(T)*S(T)}{N} & 0 \\ \frac{I(T)*S(T)}{N} & -I(T) \\ 0 & I(T) \end{bmatrix}_{3*T \text{ x } 2} \quad \begin{bmatrix} \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} \frac{dS(t)}{dt}|_{t=1} \\ \frac{dI(t)}{dt}|_{t=1} \\ \frac{dR(t)}{dt}|_{t=1} \\ \frac{dS(t)}{dt}|_{t=2} \\ \frac{dI(t)}{dt}|_{t=2} \\ \frac{dR(t)}{dt}|_{t=2} \\ . \\ . \\ \frac{dS(t)}{dt}|_{t=T} \\ \frac{dI(t)}{dt}|_{t=T} \\ \frac{dR(t)}{dt}|_{t=T} \end{bmatrix}$$

0.1.1 Jacobian

The vector valued function is

$$f(\begin{bmatrix} \beta \\ \gamma \end{bmatrix}) = \begin{bmatrix} \frac{dS(t)}{dt}|_{t=1} \\ \frac{dI(t)}{dt}|_{t=1} \\ \frac{dR(t)}{dt}|_{t=1} \\ \frac{dS(t)}{dt}|_{t=2} \\ \frac{dI(t)}{dt}|_{t=2} \\ \frac{dR(t)}{dt}|_{t=2} \\ . \\ . \\ \frac{dS(t)}{dt}|_{t=T} \\ \frac{dI(t)}{dt}|_{t=T} \\ \frac{dR(t)}{dt}|_{t=T} \end{bmatrix} = \begin{bmatrix} \frac{\beta I(1)S(1)}{N} \\ \frac{\beta I(1)S(1)}{N} - \gamma I(1) \\ \gamma I(1) \\ \frac{\beta I(2)S(2)}{N} \\ \frac{\beta I(2)S(2)}{N} - \gamma I(2) \\ \gamma I(2) \\ . \\ . \\ \frac{\beta I(T)S(T)}{N} \\ \frac{\beta I(T)S(T)}{N} - \gamma I(T) \\ \gamma I(T) \end{bmatrix} = \begin{bmatrix} f_{11} \\ f_{21} \\ f_{31} \\ f_{12} \\ f_{22} \\ f_{32} \\ . \\ . \\ f_{1T} \\ f_{2T} \\ f_{3T} \end{bmatrix}$$

Calculating the Jacobian Matrix $$ J = $$
$$
\begin{bmatrix}
\frac{df_{11}}{d\beta} & \frac{df_{11}}{d\gamma} \\
\frac{df_{21}}{d\beta} & \frac{df_{21}}{d\gamma} \\
\frac{df_{31}}{d\beta} & \frac{df_{31}}{d\gamma} \\
\frac{df_{12}}{d\beta} & \frac{df_{12}}{d\gamma} \\
\frac{df_{22}}{d\beta} & \frac{df_{22}}{d\gamma} \\
\frac{df_{32}}{d\beta} & \frac{df_{32}}{d\gamma} \\
. & \\
. & \\
\frac{df_{1T}}{d\beta} & \frac{df_{1T}}{d\gamma} \\
\frac{df_{2T}}{d\beta} & \frac{df_{2T}}{d\gamma} \\
\frac{df_{3T}}{d\beta} & \frac{df_{3T}}{d\gamma}
\end{bmatrix}
=
\begin{bmatrix}
-\frac{I(1)*S(1)}{N} & 0 \\
\frac{I(1)*S(1)}{N} & -I(1) \\
0 & I(1) \\
-\frac{I(2)*S(2)}{N} & 0 \\
\frac{I(2)*S(2)}{N} & -I(2) \\
0 & I(2) \\
. & \\
. & \\
-\frac{I(T)*S(T)}{N} & 0 \\
\frac{I(T)*S(T)}{N} & -I(T) \\
0 & I(T)
\end{bmatrix}
$$

The Jacobian matrix is equivalent to the Coefficient Matrix A which we used for the formulating the least squares problem.

[3]:
```
from tabulate import tabulate

# Jacobian Matrix Get Rank and Condition Number
def get_rank_and_condition_number(data_frame):
    J = np.array([[0.0 for i in range(2) ] for j in range(3*len(data_frame))])
    for i in range(len(data_frame)):
        current_day = data_frame.loc[i]
        J[3*i,0] = -((current_day.loc['S(t)']*current_day.loc['I(t)'])/␣
↪POPULATION)
        J[3*i+1,0] = ((current_day.loc['S(t)']*current_day.loc['I(t)'])/␣
↪POPULATION)
        J[3*i+1,1] = -current_day.loc['I(t)']
        J[3*i+2,1] = current_day.loc['I(t)']

    rank = np.linalg.matrix_rank(J)
    cn_number = np.linalg.cond(J)
    return J,rank,cn_number

jacobian, rank , cn_number = get_rank_and_condition_number(time_series_df)
```

[4]:
```
print("Rank of Jacobian Matrix = " + str(rank))
print("Condition Number of Jacobian Matrix = " + str(cn_number))
```

```
Rank of Jacobian Matrix = 2
Condition Number of Jacobian Matrix = 1.7337091591176315
```

Rank of Jacobian Matrix = 2 Conditon Number of Jacobian Matrix ≈ 1.73

0.1.2 Coefficient Matrix

Since the Coefficient matrix (A) is equivalent to the Jacobian Matrix, The Condition number of the Coefficient Matrix = Condition Number of the Jacobian

$$\kappa(\text{Jacobian}) = \kappa(\text{Coefficient Matrix})$$

Conditon Number of Coefficient Matrix ≈ 1.73

Solution to Least Squares Problem

[5]:
```
# Cholesky Factorization Alogrithm
def cholesky(A):
    n = A.shape[0]
    ANS = np.zeros((A.shape[0],A.shape[1]))
    for j in range(n):
        s = 0.0
        for k in range(j):
            s += ANS[j][k] * ANS[j][k]

        ANS[j][j] = np.sqrt(A[j][j] - s)
        for i in range(j+1,n):
            s = 0.0
            for k in range(0,j):
                s += ANS[i][k] * ANS[j][k]
            ANS[i][j] = (1.0/ANS[j][j] * (A[i][j]-s))
    return ANS

# Convert Matrix to Symmetric Matrix.
def convert_to_symmetric(X):
    return 0.5*(X + X.T)


# Back Subsitution.
def back_substitution(A,b):
    solution = np.zeros(A.shape[0])
    for i in range(A.shape[0]- 1, -1,-1):
        tmp = b[i][0]

        for j in range(A.shape[0]-1,i,-1):
            tmp -= solution[j] * A[i,j]
```

```
        solution[i] = tmp / A[i,i]
    return solution

# Forward Substitution Algorithm.
def forward_substitute(A,b):

    solution = np.zeros(A.shape[0])
    for i in range(A.shape[0]):
        tmp = b[i][0]
        for j in range(i):
            tmp -= solution[j]*A[i,j]
        solution[i] = tmp / A[i,i]
    return solution
```

[43]:
```
# Solve the linear system of equations
# Returns the solution for cholesky factorization based computation and builtin␣
 ↪function scipy.optimize.

from scipy.optimize import nnls
# Solve for beta and gamma for the data upto time t
def solve(day_param,derivative):
    '''
        day_param : A dataframe with single row having columns S(t),I(t),R(t)␣
 ↪for a single day
        derivative : ds/dt, di/dt and dr/dt for a particular day.
    '''

    num_days = len(day_param)


    A_rank = set()  # Used to store the rank of matrix A (Existence of Cholesky␣
 ↪Factorization)
    X = np.zeros((3*num_days,2))
    y = np.zeros((3*num_days,1))

    i_s_n = (day_param['S(t)'] * day_param['I(t)'])/ (POPULATION)

    for i in range(len(i_s_n)):

        X[3*i,0] = -i_s_n.loc[i]
        X[3*i+1,0] = i_s_n.loc[i]
        X[3*i + 1,1] = -day_param.loc[i,'I(t)']
        X[3*i + 2,1] = day_param.loc[i,'I(t)']

        y[3*i,0] = derivative.loc[i,'ds/dt']
        y[3*i+1,0] = derivative.loc[i,'di/dt']
        y[3*i+2,0] = derivative.loc[i,'dr/dt']
```

```
    A = np.matmul(X.T,X)
    b  = np.matmul(X.T,y)
    A_rank.add(np.linalg.matrix_rank(A))
    #A_sym = convert_to_symmetric(A)

    A_tri = cholesky(A)
    inter = forward_substitute(A_tri,b)
    inter = np.expand_dims(inter,1)

    sol = back_substitution(A_tri.T,inter)
    scipy_solution = nnls(X,y.squeeze())
    return sol,scipy_solution,A_rank
```

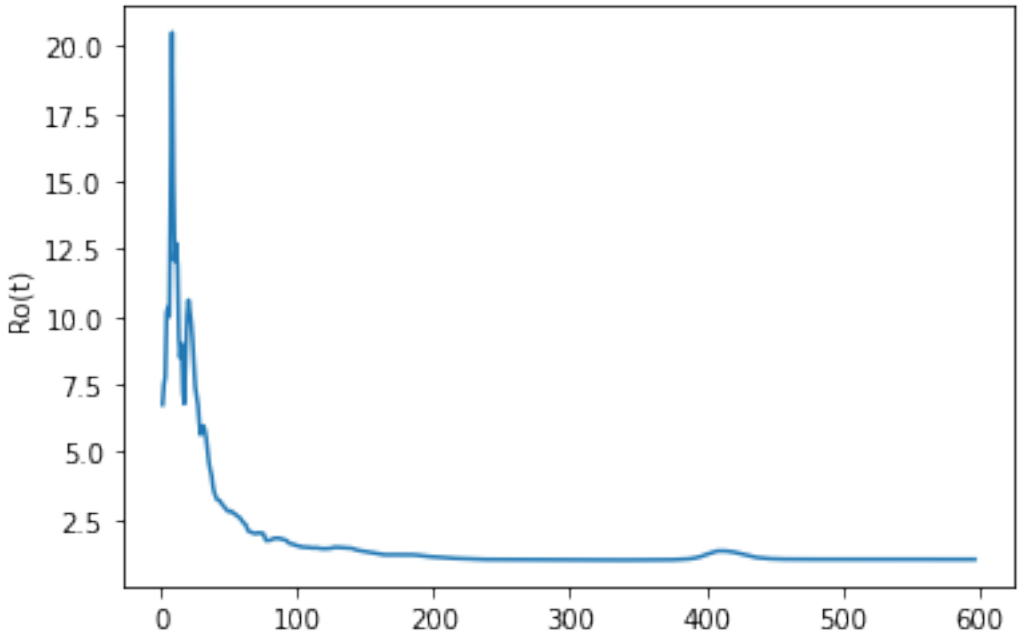
[ ]:

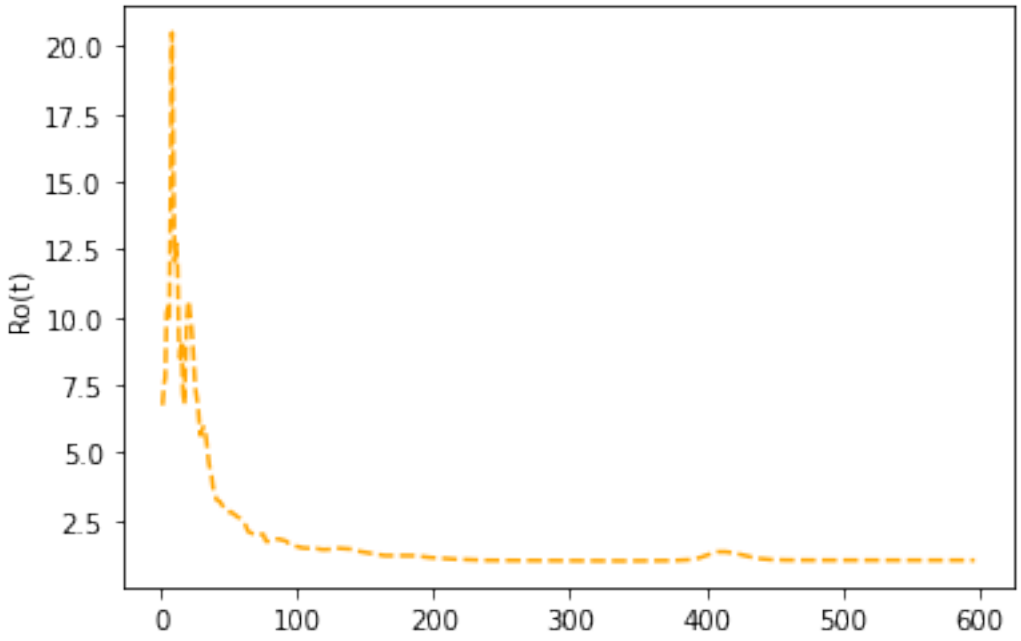
[7]:
```
# Calculate ds/dt, di/dt and dr/dt
derivative_df = pd.DataFrame(0.0,index=np.
 ↪arange(len(time_series_df)),columns=['ds/dt','di/dt','dr/dt'])

for i in range(len(time_series_df)):
    if i==0:
        derivative_df.loc[i,"ds/dt"] = (time_series_df.loc[i+1,"S(t)"] -␣
 ↪time_series_df.loc[i,"S(t)"])
        derivative_df.loc[i,"di/dt"] = (time_series_df.loc[i+1,"I(t)"] -␣
 ↪time_series_df.loc[i,"I(t)"])
        derivative_df.loc[i,"dr/dt"] = (time_series_df.loc[i+1,"R(t)"] -␣
 ↪time_series_df.loc[i,"R(t)"])
    elif i == len(time_series_df)-1:
        derivative_df.loc[i,"ds/dt"] = (time_series_df.loc[i,"S(t)"] -␣
 ↪time_series_df.loc[i-1,"S(t)"])
        derivative_df.loc[i,"di/dt"] = (time_series_df.loc[i,"I(t)"] -␣
 ↪time_series_df.loc[i-1,"I(t)"])
        derivative_df.loc[i,"dr/dt"] = (time_series_df.loc[i,"R(t)"] -␣
 ↪time_series_df.loc[i-1,"R(t)"])
    else:
        derivative_df.loc[i,"ds/dt"] = (time_series_df.loc[i+1,"S(t)"] -␣
 ↪time_series_df.loc[i-1,"S(t)"])/2.0
        derivative_df.loc[i,"di/dt"] = (time_series_df.loc[i+1,"I(t)"] -␣
 ↪time_series_df.loc[i-1,"I(t)"])/2.0
        derivative_df.loc[i,"dr/dt"] = (time_series_df.loc[i+1,"R(t)"] -␣
 ↪time_series_df.loc[i-1,"R(t)"])/2.0
```

[8]:
```
indices = []
plot_points = []
```

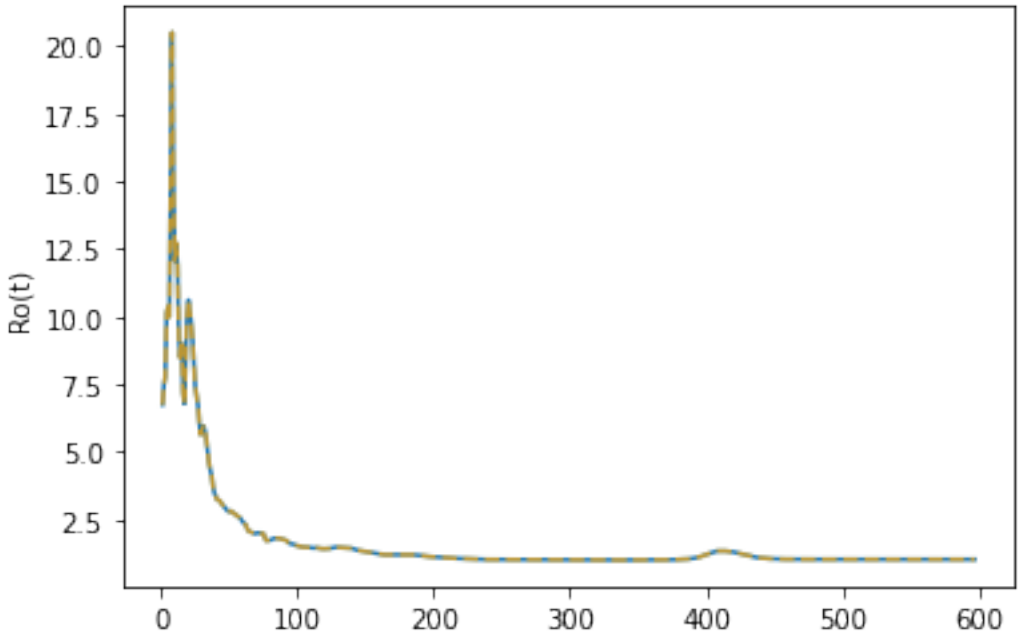
```
scipy_solutions = []
scipy_sol_error = []

# Get Ro(t)
for i in range(len(time_series_df)):
    sol,scipy_solution,MatrixA_rank = solve(time_series_df[:i+1],derivative_df[:
 ↪i+1])
    scipy_solutions.append(scipy_solution[0][0]/scipy_solution[0][1])
    scipy_sol_error.append(scipy_solution[1])
    plot_points.append(sol[0]/sol[1])
    indices.append(i+1)
```

[9]:
```
plt.ylabel('Ro(t)')
plt.plot(indices,plot_points)
plt.show()
plt.ylabel('Ro(t)')
plt.plot(indices,scipy_solutions,'--',color = 'orange',alpha=1.0)
plt.show()
```





[10]:
```
# Overlaying the plots on one another
plt.ylabel('Ro(t)')
plt.plot(indices,plot_points)
plt.plot(indices,scipy_solutions,'--',color = 'orange',alpha=0.7)
plt.show()
```



- Couldnt find any visible difference between the plots Ro(t) from the values computed using cholesky and scipy.optimize

[11]:
```
# Check if Ro(t) computed from cholesky and scipy are the same
# Computing the absolute difference between the two summing and averaging up
error = 0.0
for i in range(len(indices)):
    error += np.abs(scipy_solutions[i] - plot_points[i] )
    #print(str(scipy_solutions[i]) + "    ;;;; " + str(plot_points[i]) + ",,,,"␣
 ↪+ str(scipy_sol_error[i]))
print("Average error = " + str(error/len(indices)))
```

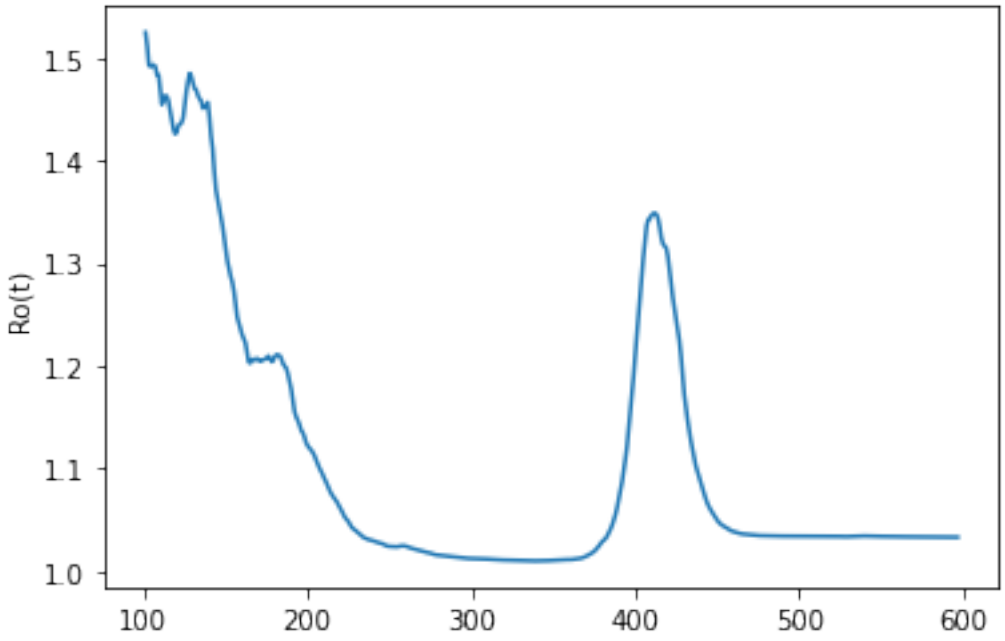
```
Average error = 1.4126053760557268e-15
```

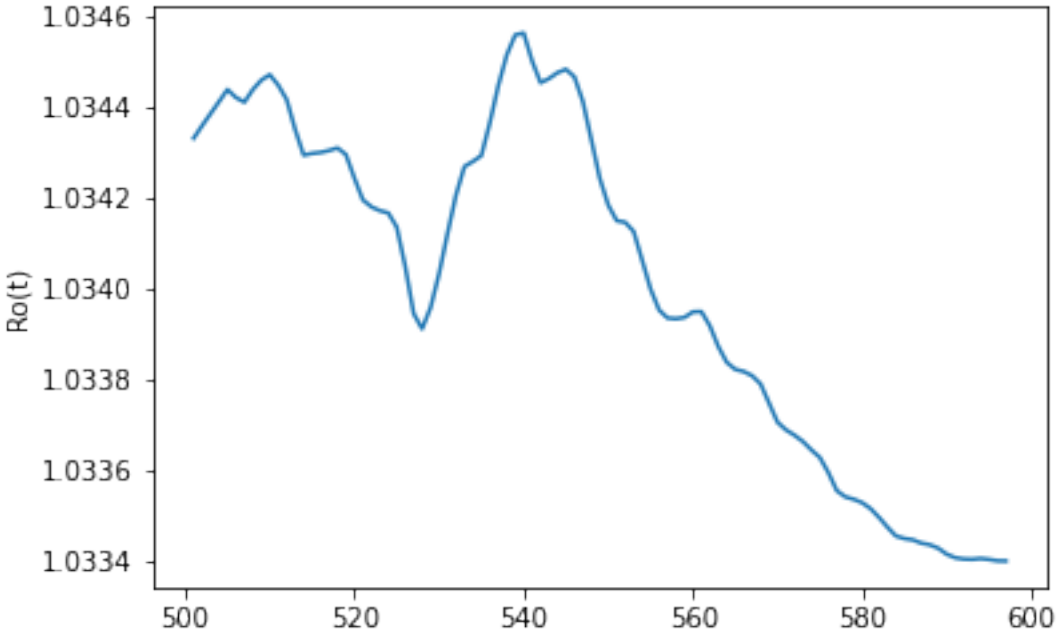
The average error is very small (10^-15). The solutions obtained using the cholesky factorization based solver and builtin solver is almost equal to each other.

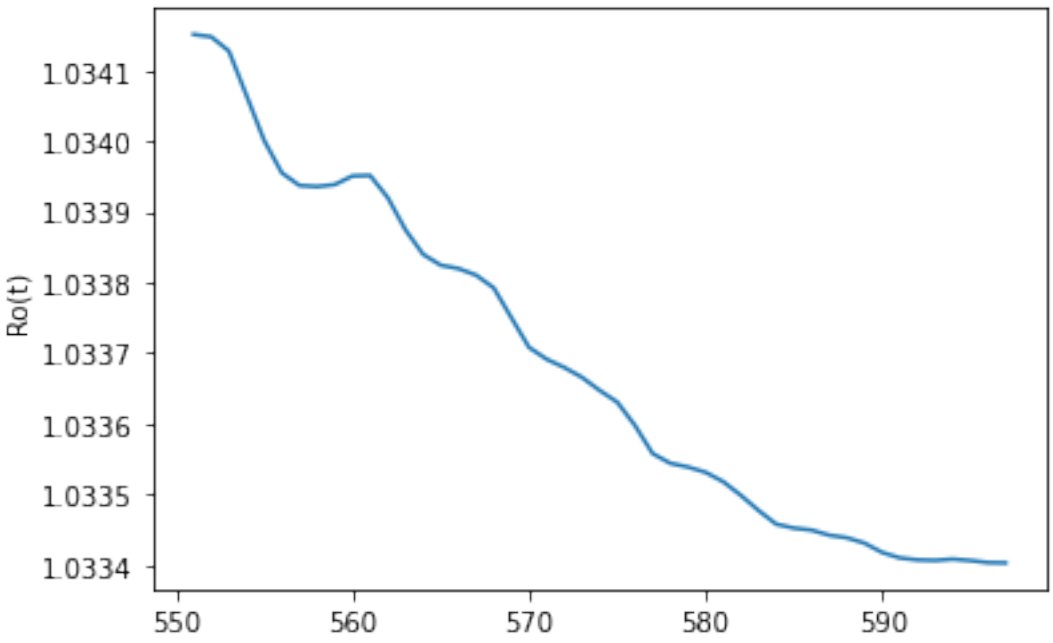
- Since the rank of the Coefficient Matrix (also equivalent to the Jacobian) is 2 which is equal to number of variables, a solution to the the linear system of equations which we formulated exist.
- The condition number of the Coefficient Matrix (≈ 1) the inverse of the Coefficient Matrix can be computed with good accuracy (Wikipedia) which might be another possible reason that the average error between the Cholesky factorization based solver and builtin solver being negligible.

Zooming into the 'Ro(t) vs t' plot from t=100 onwards and t=500 and t=550 onwards

[12]:
```
# Zooming in towards the 100 th day

plt.ylabel('Ro(t)')
plt.plot(indices[100:],plot_points[100:])
plt.show()


# Zooming in towards the 500 th day
plt.ylabel('Ro(t)')
plt.plot(indices[500:],plot_points[500:])
plt.show()

# Zomming in towards the 550th day
plt.ylabel('Ro(t)')
plt.plot(indices[550:],plot_points[550:])
plt.show()
```


Ro(t)
1.5
1.4
1.3
1.2
1.1
1.0
100
200
300
400
500
600


Ro(t)
1.0346
1.0344
1.0342
1.0340
1.0338
1.0336
1.0334
500
520
540
560
580
600



- We can see a huge peak(Ro(t) $>$ 1) in the range t = 400-500 and t=100-200 which is in agreement with the peaks in the I(t) graph. There is an exponential increase in infection spread.
- Towards the end we can see declining Ro(t) values.
- Upon manual observation of values of Ro(t) towards the end ($>$ 500 th day) we observe it is closer to 1.0 but slightly above ≈ 1.03
- We cant clearly tell whether a peak is present (≈ 1.03) but seeing the declining trend and the closeness of values towards 1.0, we can say that the peak has passed as of Oct 2020.

0.2 Question : 2

Steepest Descent and Newton's Line Search Methods

1. Find the minima x^* for the given functions $f_1(x)$ and $f_2(x)$ using your own implementation of Steepest Descent. Compute the step lenght by implementing the backtracking algorithm (Algorithm 3.1 Nocedal and Wright) with $\rho = 0.9$ and $c = 10^{-4}$. **[1.5 Points]**
2. Find the minima x^* for the given functions $f_1(x)$ and $f_2(x)$ using your own implementation of Newton's Method. **[1 Point]**

Notes: 1. Run both algorithms for two initial guesses. i. $x_0 = (2, 0)$ and ii. $x_0 = (2, 2)$ 2. Stop iterations when $||x_{k+1} - x_k||_2^2 < 10^{-5}$ 3. For each case report the solution and the number of iterations to converge. Also comment on the reported number of iterations. 4. Show the function contour plot and the iterates $\{x_k\}$ including the solution.

Consider the following quadratic functions: 1. \$f_1(x) = 1$\frac{}{2xGTA_1x\$}$

where $A_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

2. $f_2(x) = 1_{\overline{2xTA\_2x}}$

   where $A_2 = \begin{pmatrix} 10 & 8 \\ 8 & 10 \end{pmatrix}$

[13]:
```
# Some Helper functions
def squared_norm(x):
    x_val = np.float64(x[0][0])
    y_val = np.float64(x[1][0])
    return x_val*x_val + y_val*y_val

def norm(x):
    return np.sqrt(squared_norm(x))

def get_unit_vector(x):
    norm_value = norm(x)
    if norm_value == np.float64(0.0):
        return x
    return (x/norm_value).copy()
```

[14]:
```
# Definition of the functions,
# The Jacobians  and the hessian

def f1(x,y):

    # x,y  = Elements of 2*1 vector
    vec = np.array([[x],[y]])
    A = np.array([[1,0],[0,1]])
    h = np.matmul(vec.T,A)
    return 0.5*np.matmul(h,vec)

def f2(x,y):

    # x,y  = Elements of 2*1 vector
    vec = np.array([[x],[y]])
    A = np.array([[10,8],[8,10]])
    h = np.matmul(vec.T,A)
    return 0.5*np.matmul(h,vec)


def gradient_f1(x):

    # x = Input 2*1 vector
    x_value = np.float64(x[0][0])
    y_value = np.float64(x[1][0])
```

```
    return np.array([[x_value],[y_value]])

def gradient_f2(x):
    # x =  2*1 vector
    x_value = np.float64(x[0][0])
    y_value = np.float64(x[1][0])
    return np.array([[10*x_value + 8 *y_value],[10*y_value + 8 * x_value]])

def hessian_f1(x):
    # x =  2*1 vector
    return np.array([[1,0],[0,1]])

def hessian_f2(x):
    # x =  2*1 vector
    return np.array([[10,8],[8,10]])
```

[15]:
```
def steepest_descent(initial_point,function='f1'):
    '''
        Arguments
            - initial_point = 2*1 vector specifying the initial point.
            - function = String to refer to the function which needs to be␣
 ↪optimized.

    '''
    fun = f1 if function == 'f1' else f2
    gradient_fun = gradient_f1 if function == 'f1' else gradient_f2

    current_point = initial_point.copy()
    c = 0.0001
    rho = 0.9
    iterations = 0
    iterates = []

    iterates.append(current_point)
    while True:

        iterations += 1
        gradient = gradient_fun(current_point)
        # Direction of p should be opposite to the direction of the gradient
        p = -get_unit_vector(gradient)

        # Intial step size
        alpha  = np.float64(1.0)
        previous_point = current_point.copy()


        break_inner_loop = False
```

```
        current_x = np.float64(current_point[0][0])
        current_y = np.float64(current_point[1][0])
        current_function_value = np.float64(fun(current_x,current_y).squeeze())


        while not break_inner_loop:

            new_point = current_point + alpha*p

            new_x = np.float64(new_point[0][0])
            new_y = np.float64(new_point[1][0])

            # Calculate the terms in the Taylor Series Approximation.
            new_function_value = np.float64(fun(new_x,new_y).squeeze())
            increment_value = np.float64(((c*alpha)*np.dot(gradient.T,p)).
↪squeeze())

            # Decreasing the step size until sufficient decrease
            if((new_function_value > (current_function_value +␣
↪increment_value))):
                alpha = alpha * rho
            else:
                break_inner_loop = True


        # Perform descent using the step size computed
        new_point = current_point + alpha*p
        current_point = new_point.copy()
        iterates.append(current_point)
        # Condition for stopping
        if np.linalg.norm(current_point-previous_point) < 0.00001:
            break
    return iterations,iterates
```

[16]:
```
def newtons_method(initial_point,function = 'f1'):

    '''
        Arguments
            - initial_point = 2*1 vector specifying the initial point.
            - function = String to refer to the function which needs to be␣
↪optimized.

    '''

    fun = f1 if function == 'f1' else f2
    gradient_fun = gradient_f1 if function == 'f1' else gradient_f2
```

```
    hessian_fun = hessian_f1 if function =='f1' else hessian_f2    # Required for␣
↪calculating the step's direction


    current_point = initial_point.copy()
    iterations = 0
    iterates = []
    iterates.append(current_point)


    # For performing the line search
    c = 0.0001
    rho = 0.9

    while True:
        iterations+=1
        gradient = gradient_fun(current_point)
        hessian = hessian_fun(current_point)

        # Step Direction in Newtons Method
        m = np.matmul(np.linalg.inv(hessian),gradient)
        p = -m

        break_inner_loop = False

        # Initial Step Size
        alpha  = np.float64(1.0)

        current_x = np.float64(current_point[0][0])
        current_y = np.float64(current_point[1][0])
        current_function_value = np.float64(fun(current_x,current_y).squeeze())

        while not break_inner_loop:
            new_point = current_point + alpha*p
            new_x = np.float64(new_point[0][0])
            new_y = np.float64(new_point[1][0])


            # Calculating the taylor Series Approximation
            new_function_value = np.float64(fun(new_x,new_y).squeeze())
            increment_value = np.float64(((c*alpha)*np.dot(gradient.T,p)).
↪squeeze())
            second_order_term = (c*(alpha*alpha/2)*(np.matmul(np.matmul(p.
↪T,hessian),p))).squeeze()
            increment_value += np.float64(second_order_term)
```

```
            # Determining the  appropirate stp size
            if((new_function_value > (current_function_value +␣
↪increment_value))):
                alpha = alpha * rho
            else:
                break_inner_loop = True


        previous_point = current_point.copy()
        # Perform the descent
        new_point = current_point + alpha*p
        current_point = new_point.copy()
        iterates.append(current_point)
        if np.linalg.norm(current_point-previous_point) < 0.00001:
            break

    return iterations,iterates
```

[17]:
```
# Plot the Contour Plots
def plot_contour(function,iterations,iterates,title = 'Contour Plot'):

    if function == 'f1':
        fun = f1
    else:
        fun = f2

    res = 1000
    x_values = np.linspace(2.1,-2.1,res)
    y_values = np.linspace(2.1,-2.1,res)
    X,Y = np.meshgrid(x_values,y_values)

    Z = np.zeros((res,res))
    for i in range(len(X[0])):
        for j in range(len(Y[:,0])):
            z_value = fun(X[0,i],Y[j,0])
            Z[i][j] = z_value

    fig,ax=plt.subplots(1,1)
    cp = ax.contourf(X, Y, Z)
    fig.colorbar(cp)
    ax.set_title(title)
    plt.plot(iterates[0][0,0],iterates[0][1,0],'ro')
    for i in range(1,len(iterates)):
        plt.arrow(iterates[i-1][0,0],iterates[i-1][1,0],dx = iterates[i][0,0] -␣
↪iterates[i-1][0,0] ,dy = iterates[i][1,0] - iterates[i-1][1,0],width=0.
↪01,head_width = 0.09)
```

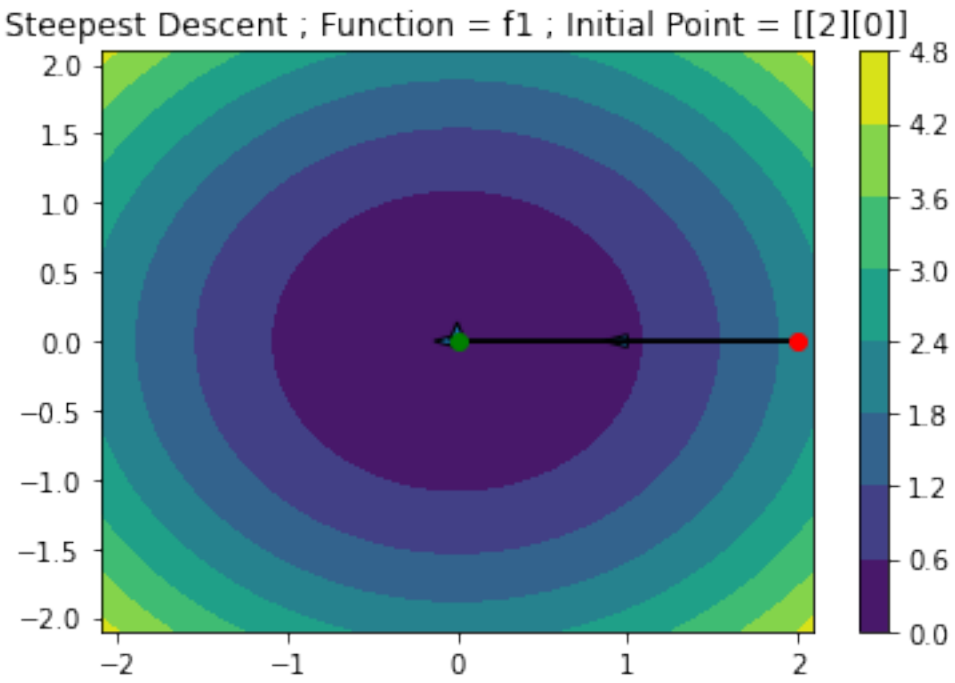
```
    plt.plot(iterates[len(iterates)-1][0,0],␣
↪iterates[len(iterates)-1][1,0],'go',linewidth=0.1)
    plt.show()
```

[ ]:

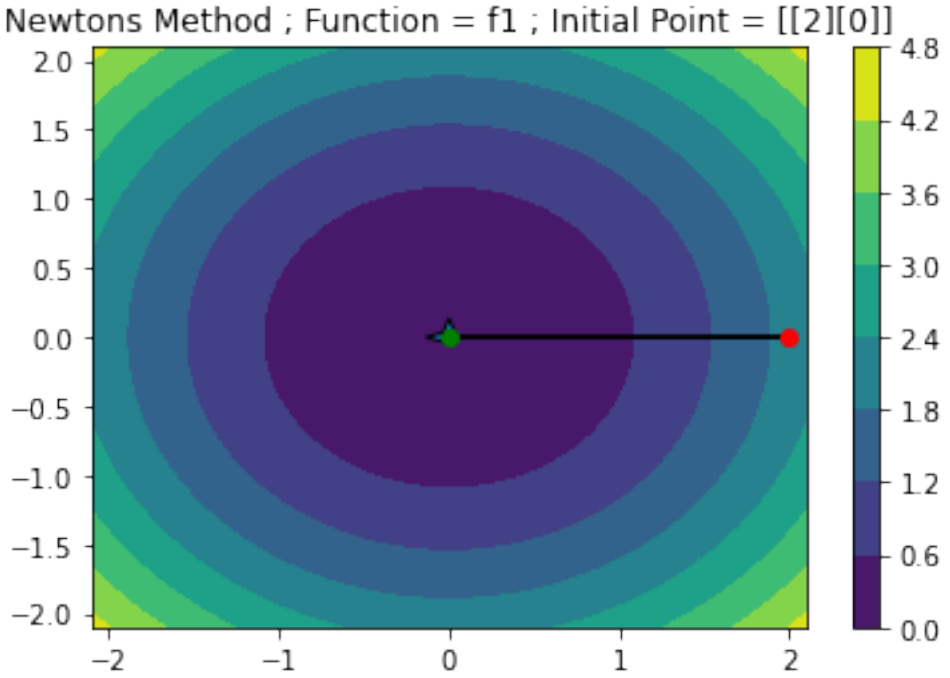
Function f1(x)

[18]:
```
# Red Dot Stands for the starting Point
# Green Dot Stands for the ending point


initial_point = np.array([[2],[0]])

print("Initial Point = [[2],[0]]")
print("----------------------------")
iterations_steep_f1 , iterates_steep_f1 = steepest_descent(initial_point,'f1')
iterations_newton_f1 , iterates_newton_f1 = newtons_method(initial_point,'f1')
print("Steepest Descent Solution = " + str(iterates_steep_f1[-1]))
print("Iterations taken for Steepest Descent= " + str(iterations_steep_f1))
plot_contour('f1',iterations_steep_f1,iterates_steep_f1,title='Steepest Descent ;
↪ Function = f1 ; Initial Point = [[2][0]]' )
print("Newtons Method Solution = " + str(iterates_newton_f1[-1]))
print("Iterations taken for Newtons Method = " + str(iterations_newton_f1))
plot_contour('f1',iterations_newton_f1,iterates_newton_f1,title='Newtons Method ;
↪ Function = f1 ; Initial Point = [[2][0]]')
```

```
Initial Point = [[2],[0]]
----------------------------
Steepest Descent Solution = [[0.]
 [0.]]
Iterations taken for Steepest Descent= 3
```

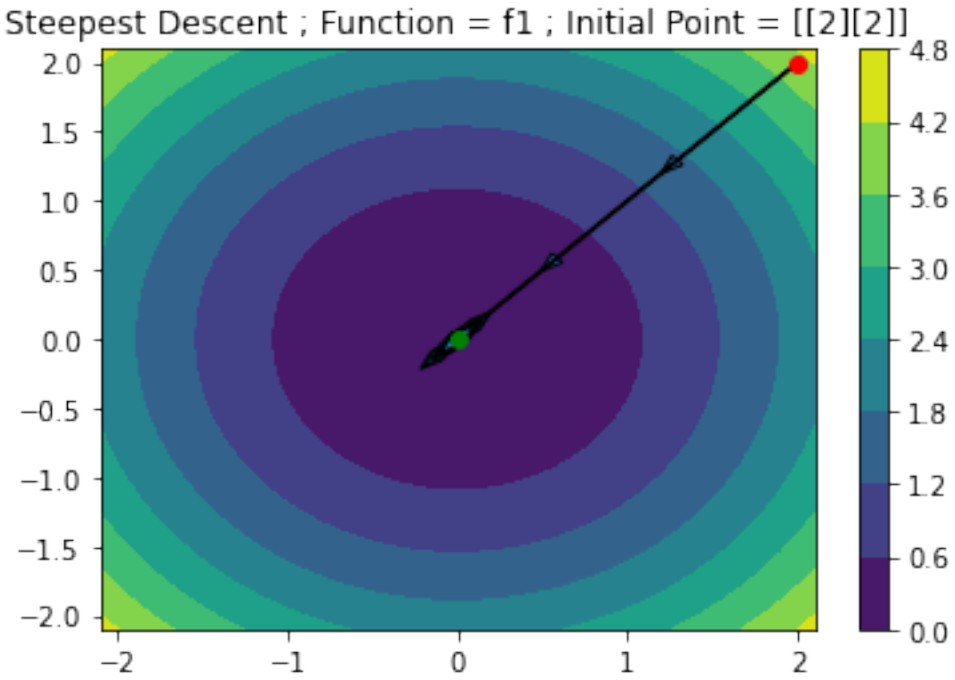



```
Newtons Method Solution = [[0.]
 [0.]]
Iterations taken for Newtons Method = 2
```

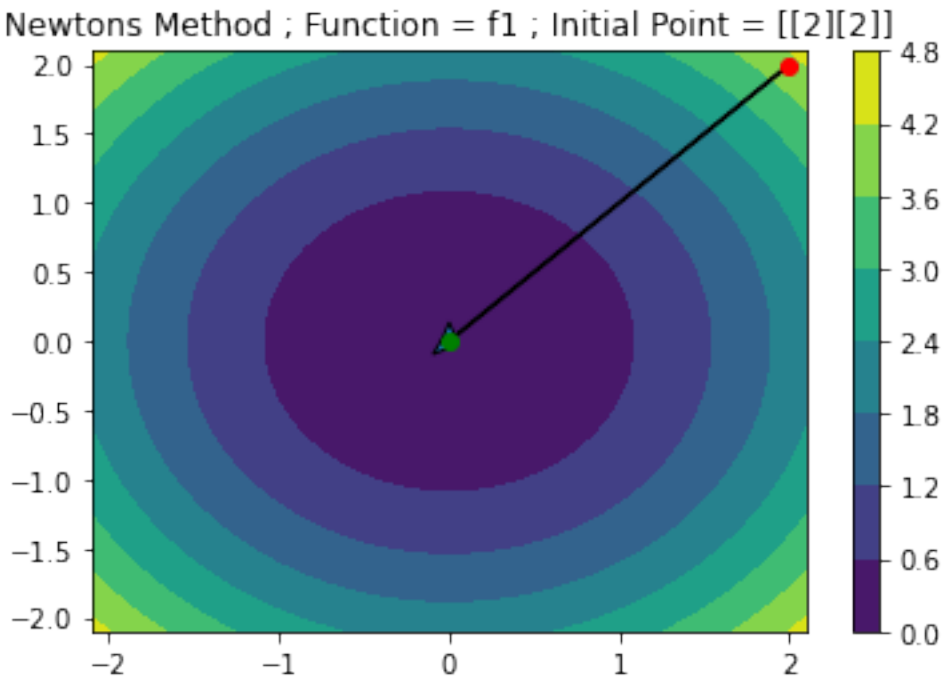


[19]:
```
initial_point = np.array([[2],[2]])

print("Initial Point = [[2],[2]]")
print("----------------------------")
iterations_steep_f1 , iterates_steep_f1 = steepest_descent(initial_point,'f1')
iterations_newton_f1 , iterates_newton_f1 = newtons_method(initial_point,'f1')
print("Steepest Descent Solution = " + str(iterates_steep_f1[-1]))
print("Iterations taken for Steepest Descent= " + str(iterations_steep_f1))
plot_contour('f1',iterations_steep_f1,iterates_steep_f1,title='Steepest Descent ;
 ↪ Function = f1 ; Initial Point = [[2][2]]' )
print("Newtons Method Solution = " + str(iterates_newton_f1[-1]))
print("Iterations taken for Newtons Method =" + str(iterations_newton_f1))
plot_contour('f1',iterations_newton_f1,iterates_newton_f1,title='Newtons Method ;
 ↪ Function = f1 ; Initial Point = [[2][2]]')
```

```
Initial Point = [[2],[2]]
----------------------------
Steepest Descent Solution = [[2.62421098e-06]
 [2.62421098e-06]]
Iterations taken for Steepest Descent= 72
```



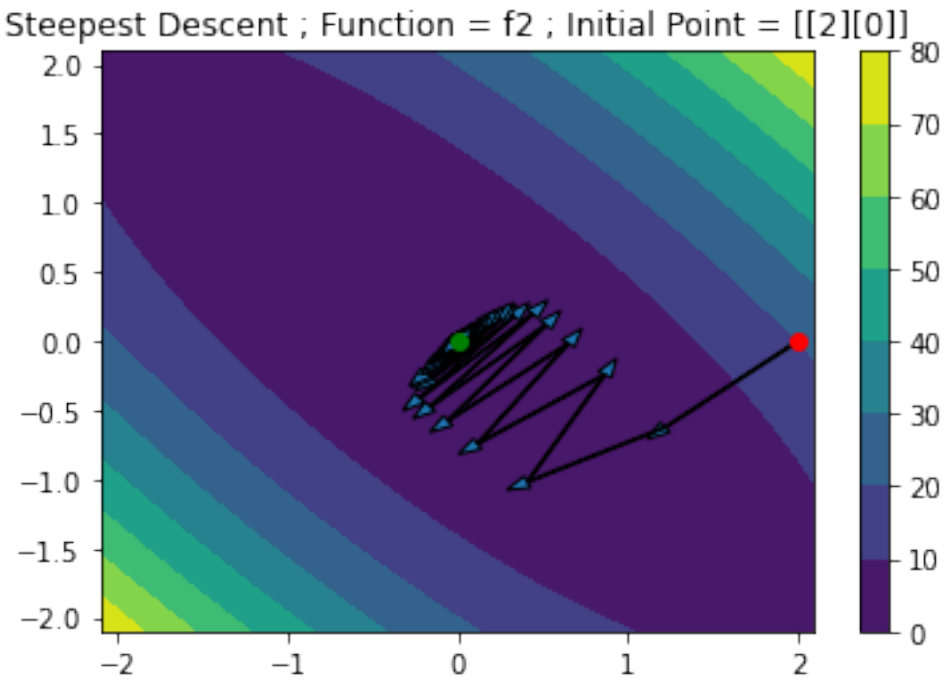

```
Newtons Method Solution = [[0.]
 [0.]]
Iterations taken for Newtons Method =2
```



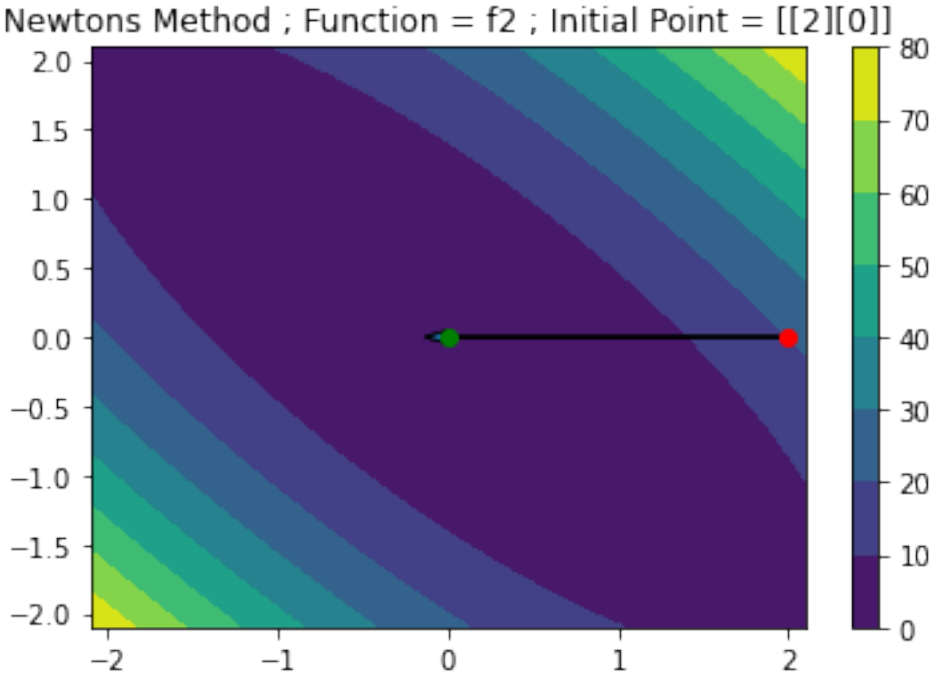
Function f2(x)

[20]:
```
initial_point = np.array([[2],[0]])

print("Initial Point = [[2],[0]]")
print("----------------------------")
iterations_steep_f2 , iterates_steep_f2 = steepest_descent(initial_point,'f2')
iterations_newton_f2 , iterates_newton_f2 = newtons_method(initial_point,'f2')
print("Steepest Descent Solution = " + str(iterates_steep_f2[-1]))
print("Iterations taken for Steepest Descent= " + str(iterations_steep_f2))
plot_contour('f2',iterations_steep_f2,iterates_steep_f2,title='Steepest Descent ;
 ↪ Function = f2 ; Initial Point = [[2][0]]' )
print("Newtons Method Solution = " + str(iterates_newton_f2[-1]))
print("Iterations taken for Newtons Method =" + str(iterations_newton_f2))
plot_contour('f2',iterations_newton_f2,iterates_newton_f2,title='Newtons Method ;
 ↪ Function = f2 ; Initial Point = [[2][0]]')
```

```
Initial Point = [[2],[0]]
----------------------------
Steepest Descent Solution = [[-2.91584190e-06]
 [-2.91738507e-06]]
Iterations taken for Steepest Descent= 90
```

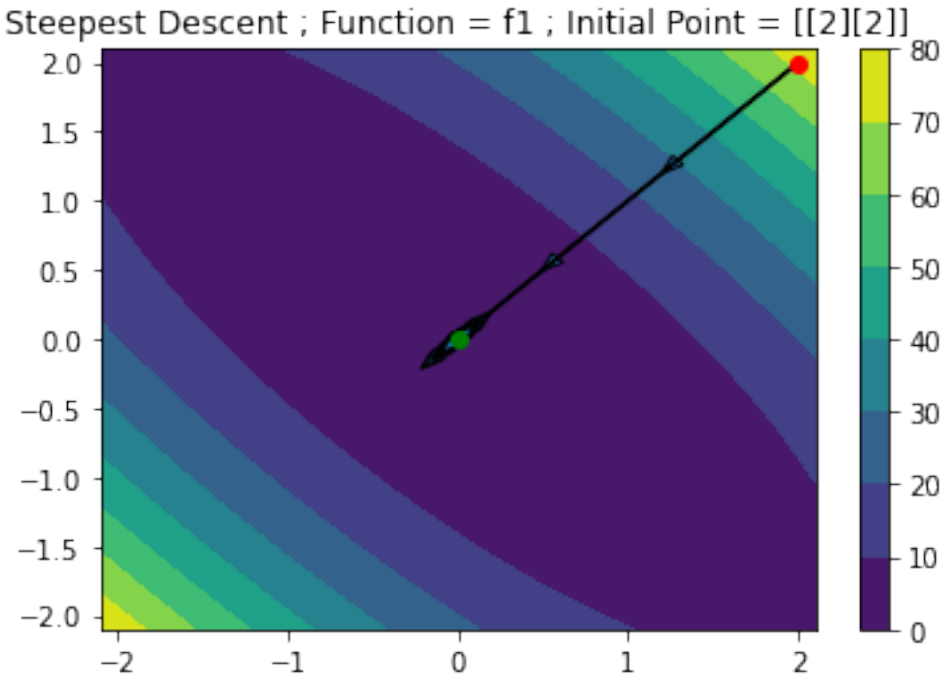



```
Newtons Method Solution = [[9.86076132e-32]
 [0.00000000e+00]]
Iterations taken for Newtons Method =2
```

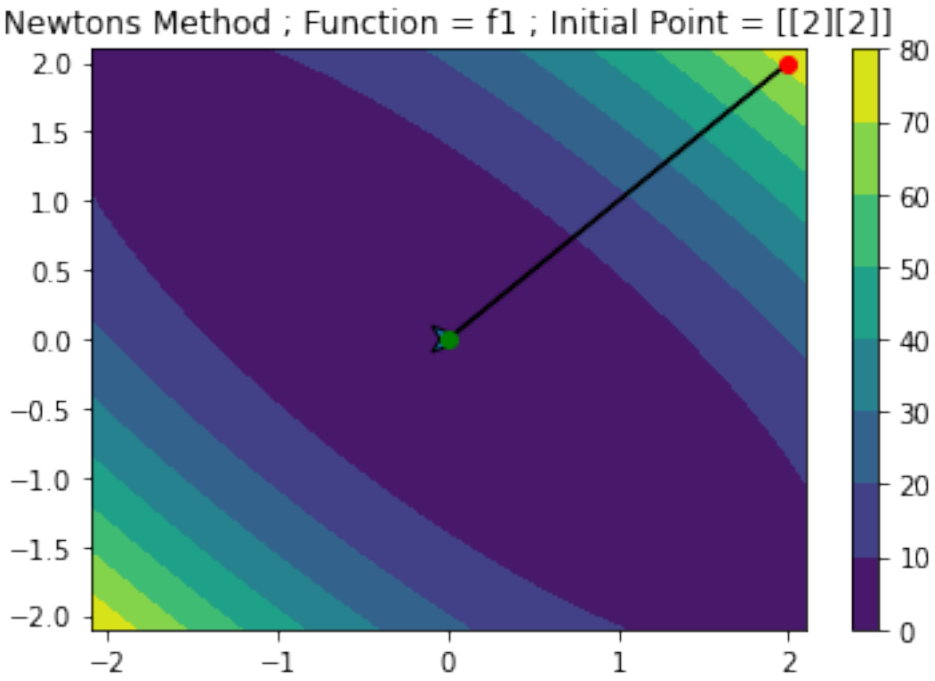


[21]:
```
initial_point = np.array([[2],[2]])

print("Initial Point = [[2],[2]]")
print("----------------------------")
iterations_steep_f2 , iterates_steep_f2 = steepest_descent(initial_point,'f2')
iterations_newton_f2 , iterates_newton_f2 = newtons_method(initial_point,'f2')
print("Steepest Descent Solution = " + str(iterates_steep_f2[-1]))
print("Iterations taken for Steepest Descent= " + str(iterations_steep_f2))
plot_contour('f2',iterations_steep_f2,iterates_steep_f2,title='Steepest Descent ;
 ↪ Function = f1 ; Initial Point = [[2][2]]' )
print("Newtons Method Solution = " + str(iterates_newton_f2[-1]))
print("Iterations taken for Newtons Method =" + str(iterations_newton_f2))
plot_contour('f2',iterations_newton_f2,iterates_newton_f2,title='Newtons Method ;
 ↪ Function = f1 ; Initial Point = [[2][2]]')
```

```
Initial Point = [[2],[2]]
----------------------------
Steepest Descent Solution = [[2.62421098e-06]
 [2.62421098e-06]]
Iterations taken for Steepest Descent= 72
```




```
Newtons Method Solution = [[0.]
 [0.]]
Iterations taken for Newtons Method =2
```



$$ f_1(x) = 1_{2xGTA_1x}

A_1 =

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$

Initial Point	Iterations(Steepest Descent)	Minima(Steepest Descent)	Iterations (Newtons Method)	Minima (Newtons Method)
(2,2)	72	$(2*10^{-6}, 2*10^{-6})$	2	(0.0,0.0)
(2,0)	3	(0.0,0.0)	2	(0.0,0.0)

$$f_2(x) = 1_{2xGTA_2x}

A_2 =

$$\begin{pmatrix} 10 & 8 \\ 8 & 10 \end{pmatrix}$$

$$

Initial Point	Iterations(Steepest Descent)	Minima(Steepest Descent)	Iterations (Newtons Method)	Minima (Newtons Method)
(2,2)	72	$(2*10^{-6}, 2*10^{-6})$	2	(0.0,0.0)
(2,0)	90	$(-2*10^{-6}, -2*10^{-6})$	2	$(9*10^{-32}$,0.0)

0.3 Observations

- Newtons Method took only two iterations to converge to a solution (One Iteration to move and one iteration to check condition) in all the cases

$$f = \frac{1}{2}x^T Ax$$

$$ \nabla f = Ax\\\

$$\nabla^2 f = A$$

The Newtons method update the parameters as $x_1 = x_0 - \alpha(\nabla^2 f)^{-1}\nabla f$. When $alpha = 1$ and substituting for $\nabla^2 f$ and ∇f

$$x_1 = x_0 - A^{-1}(Ax_0)$$

Since both A's given are symmetric positive definite inverse of A exist. So

$$x_1 = x_0 - x_0 = \vec{0}$$

We have started with initial step size as 1 and indeed it provides a decrease in function value on substitution . Therefore we can see that Newtons method is able to converge in two iterations.

- The steepest descent method takes the most amount of time compared to Newtons Method

- As we reach closer and closer to the minima the step sizes to take becomes smaller and smaller giving rise to more iterations to converge.
- When the initial point is along an eigen vector of A, We can see that the steepest method follows an almost Straight path to the minima (Inital point (2,2) in both functions $f1$ and $f2$ and Initial point (2,0) in function $f1$). We could see the steepest descent following a zig zag path for the initial point (2,0) in $f2$ (Blindly following descent direction).

[ ]:

1 Question : 3

Rosenbrock function

Use the steepest descent and Newtons algorithms using the backtracking line search to minimize the classic Rosenbrock function. Set the initial step length to 1. At each iteration store the step lengths used by each method and make plots. Show the step lengths taken and iterates as plots. Do these for a start point of search $x_0 = [1.2, 1.2]^T$ and then for the starting point $x_0 = [-1.2, 1]^T$ **[1 Point]**

1. Plot the convergence of the iterates and the objective function value. Evaluate the rate of convergence. **[0.5 Points]**
2. Call built-in functions for steepest descent and newton's method, and show the results for the above. Compare and evaluate your program. Compare the run-time of your program and built-in function. Is there a difference? Why or why not? **Hint**: Jacobians! **[1 Point]**

[22]:
```
from scipy.optimize import rosen,rosen_der
from scipy.optimize import line_search
from scipy.optimize import minimize
from scipy.optimize import approx_fprime
```

[23]:
```
# Define all the functions needed.
def rosenbrock(point):
    a = 1.0
    b = 100.0
    x = point[0]
    y = point[1]
    result = (1-x)**2
    result += b*((y - x**2)**2)
    return result

# Jacobian of the RosenBrock Function
def rosenbrock_jacobian(point):
    a = 1.0
    b = 100.0
    x = point[0]
    y = point[1]
    result = np.zeros((2,))
    result[0] = -2.0*(a-x) - 4*b*x*(y - x**2)
    result[1] = 2.0*b*(y - x**2)
    return result

# Hessian of the RosenBrock Function
def rosenbrock_hessian(point):
    a = 1.0
    b = 100.0
    x = point[0]
    y = point[1]
```

```
    hessian = np.zeros((2,2))
    hessian[0][0] = 2+12*b*x*x - 4*b*y
    hessian[0][1] = -4*b*x
    hessian[1][0] = -4*b*x
    hessian[1][1] = 2*b
    return hessian
```

[24]:
```
def steepest_descent_rosenbrock(point, no_stopping_criterion=False,␣
↪return_extra_info = True):
    # No stopping criterion is used for collecting enough number of samples to␣
↪evaluate the rate of converge
    # We are doing steepest descent on the RosenBrock Function
    fun = rosenbrock
    gradient_fun = rosenbrock_jacobian

    # Get a copy of the initial point
    current_point = point.copy()

    # Parameters for the steepest descent
    c = 0.0001
    rho = 0.9

    # The parameters to be returned
    iterations = 0

    if return_extra_info:
        step_sizes = []
        iterates = []
        function_values = []

        iterates.append(current_point)

        function_values.append(np.float64(fun(current_point.squeeze()).
↪squeeze()))

    while True:
        iterations += 1

        # Calculate the descent direction.
        gradient = gradient_fun(current_point)
        gradient = np.expand_dims(gradient,axis=1)
        p = -get_unit_vector(gradient)

        # Initial Step Size
        alpha  = np.float64(1.0)
        previous_point = current_point.copy()
        break_inner_loop = False
```

```
        # Function value at the current point.
        current_function_value = np.float64(fun(current_point.squeeze()).
↪squeeze())

        # BackTracking Alogrithm.
        while not break_inner_loop:

            new_point = current_point + alpha*p

            new_function_value = np.float64(fun(new_point.squeeze()).squeeze())
            increment_value = np.float64(((c*alpha)*np.dot(gradient.T,p)).
↪squeeze())

            if((new_function_value > (current_function_value +␣
↪increment_value))):
                alpha = alpha * rho
            else:
                break_inner_loop = True

        # Perform descent
        new_point = current_point + alpha*p
        current_point = new_point.copy()

        if return_extra_info:
            step_sizes.append(alpha)

            iterates.append(current_point)
            function_values.append(np.float64(fun(new_point.squeeze()).
↪squeeze()))

        # Stopping Criterion
        if not no_stopping_criterion:
            if np.linalg.norm(current_point-previous_point) < 0.00001:
                break
        else:
            if(iterations >= 3000):
              break
    if return_extra_info:
        return iterations,iterates, function_values, step_sizes
    else:
        return iterations,current_point
```

[25]:
```
def newton_rosenbrock(point, no_stopping_criterion = False,return_extra_info =␣
↪True):
```

```
    # No stopping criterion is used for collecting enough number of samples to␣
↪evaluate the rate of converge
    # We are doing Newtons Method on the RosenBrock Function
    fun = rosenbrock
    gradient_fun = rosenbrock_jacobian
    hessian_fun = rosenbrock_hessian


    current_point = point.copy()

    # The items which needs to be returned.
    iterations = 0

    if return_extra_info:
        function_values = []
        iterates = []
        step_sizes = []

        iterates.append(current_point)
        function_values.append(np.float64(fun(current_point.squeeze()).
↪squeeze()))

    # Parameters for determining the step length.
    c = 0.0001
    rho = 0.9

    while True:

        iterations+=1
        gradient = np.expand_dims(gradient_fun(current_point.squeeze()),axis=1)
        hessian = hessian_fun(current_point.squeeze())

        # The Step Direction
        m = np.matmul(np.linalg.inv(hessian),gradient)
        p = -m

        break_inner_loop = False
        alpha  = np.float64(1.0)
        current_function_value = np.float64(fun(current_point.squeeze()).
↪squeeze())

       # Backtracking search to determine the step length.
        while not break_inner_loop:

            new_point = current_point + alpha*p
            new_function_value = np.float64(fun(new_point.squeeze()).squeeze())
```

```
            increment_value = np.float64(((c*alpha)*np.dot(gradient.T,p)).
↪squeeze())
            second_order_term = (c*(alpha*alpha/2)*(np.matmul(np.matmul(p.
↪T,hessian),p))).squeeze()
            increment_value += np.float64(second_order_term)


            if((new_function_value > (current_function_value +␣
↪increment_value))):
                alpha = alpha * rho
            else:
                break_inner_loop = True


        previous_point = current_point.copy()
        new_point = current_point + alpha*p
        current_point = new_point.copy()

        if return_extra_info:
            step_sizes.append(alpha)
            function_values.append(np.float64(fun(new_point.squeeze()).
↪squeeze()))
            iterates.append(current_point)


        # Stopping Condition
        if not no_stopping_criterion:
            if np.linalg.norm(current_point-previous_point) < 0.00001:
                break
        else:
            if (iterations >= 3000):
                break

    if return_extra_info:
        return iterations,iterates,function_values,step_sizes
    else:
        return iterations,current_point
```

[26]:
```
# Helper Function for plotting the step length as a function of time and␣
↪iterates
def␣
↪plot_steplength_iterates(step_sizes,iterates,lower_limit,upper_limit,title1='Contour␣
↪plot',title2 = 'Step Size'):


    res = 3000
    x_values = np.linspace(lower_limit,upper_limit,res)
```

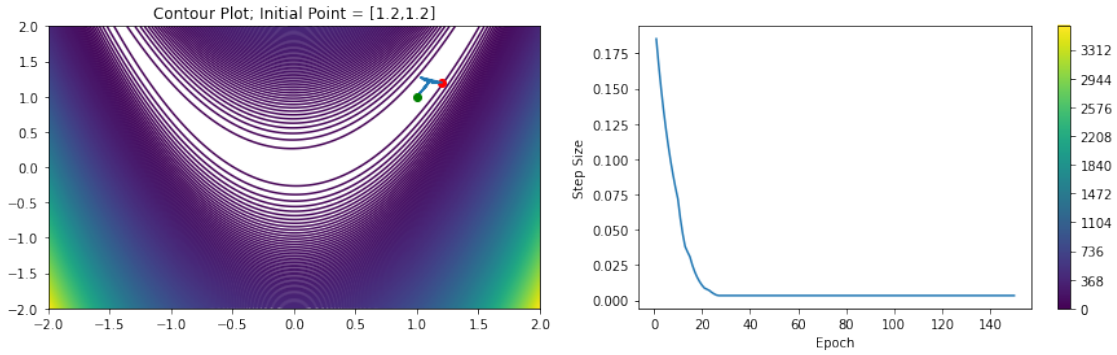
```
    y_values = np.linspace(lower_limit,upper_limit,res)
    X,Y = np.meshgrid(x_values,y_values)

    plot_fn = lambda x,y : rosenbrock(np.array([x,y]))
    Z = plot_fn(X,Y)


    fig,ax=plt.subplots(1,2)
    fig.set_figwidth(15)
    cp = ax[0].contour(X, Y, Z,500)
    fig.colorbar(cp)
    ax[0].set_title(title1)

    ax[0].plot(iterates[0,0,0],iterates[0,1,0],'ro')
    ax[0].plot(iterates[:,0,0],iterates[:,1,0])
    ax[0].plot(iterates[len(iterates)-1,0,0], iterates[len(iterates)-1,1,0],'go')

    ax[1].plot(np.arange(1,len(iterates))[:150],step_sizes[:150])
    ax[1].set_xlabel('Epoch')
    ax[1].set_ylabel(title2)
    plt.show()
```

Initial Point (1.2,1.2)

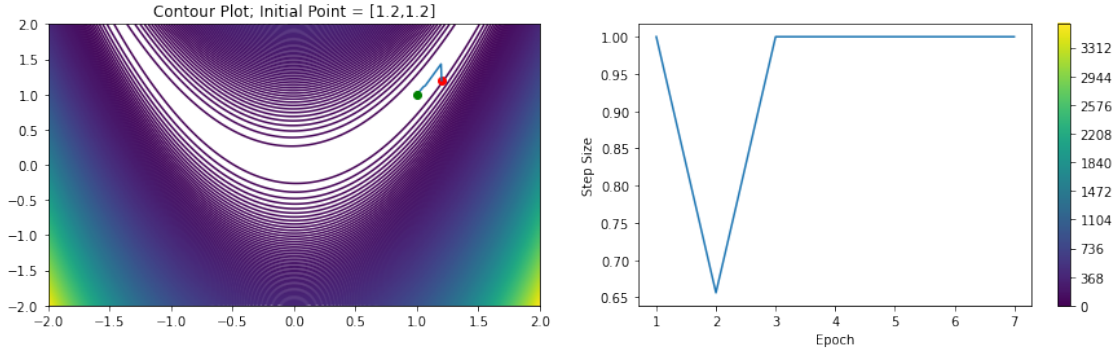
[27]:
```
initial_point = np.array([[1.2],[1.2]])
iterations_steep_p1, iterates_steep_p1, function_values_steep_p1,␣
↪step_sizes_steep_p1 = steepest_descent_rosenbrock(initial_point)
iterations_newton_p1, iterates_newton_p1, function_values_newton_p1,␣
↪step_sizes_newton_p1 = newton_rosenbrock(initial_point)
```

[28]:
```
# Red Dot Stands for the starting Point
# Green Dot Stands for the ending point

print ("Steepest Descent Iterates and Step sizes")
plot_steplength_iterates(step_sizes_steep_p1,np.array(iterates_steep_p1),-2.0,2.
↪0,title1='Contour Plot; Initial Point = [1.2,1.2]')
print ("Newtons Method Iterates and Step sizes")
plot_steplength_iterates(step_sizes_newton_p1,np.array(iterates_newton_p1),-2.
↪0,2.0,title1='Contour Plot; Initial Point = [1.2,1.2]')
```

```
Steepest Descent Iterates and Step sizes

/var/folders/sm/_xvm618541nc11hb7mv_kpj40000gn/T/ipykernel_9281/1545082446.py:17
: MatplotlibDeprecationWarning: Starting from Matplotlib 3.6, colorbar() will
steal space from the mappable's axes, rather than from the current axes, to
place the colorbar.  To silence this warning, explicitly pass the 'ax' argument
to colorbar().
  fig.colorbar(cp)
```



Newtons Method Iterates and Step sizes



Initial Point = [-1.2,1]

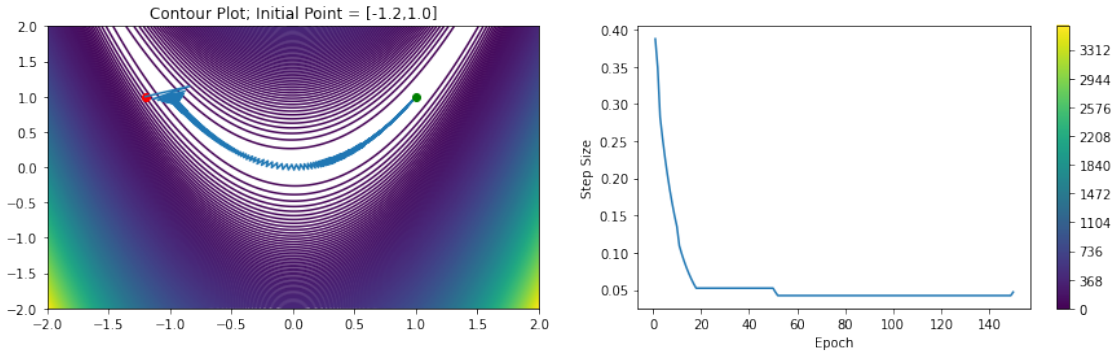
[29]:
```
initial_point = np.array([[-1.2],[1.0]])
iterations_steep_p2, iterates_steep_p2, function_values_steep_p2,␣
↪step_sizes_steep_p2 = steepest_descent_rosenbrock(initial_point)
iterations_newton_p2, iterates_newton_p2, function_values_newton_p2,␣
↪step_sizes_newton_p2 = newton_rosenbrock(initial_point)
```

[30]:
```
print ("Steepest Descent Iterates and Step sizes")
plot_steplength_iterates(step_sizes_steep_p2,np.array(iterates_steep_p2),-2.0,2.
↪0,title1='Contour Plot; Initial Point = [-1.2,1.0]')
print ("Newtons Method Iterates and Step sizes")
plot_steplength_iterates(step_sizes_newton_p2,np.array(iterates_newton_p2),-2.
↪0,2.0,title1='Contour Plot; Initial Point = [-1.2,1.0]')
```

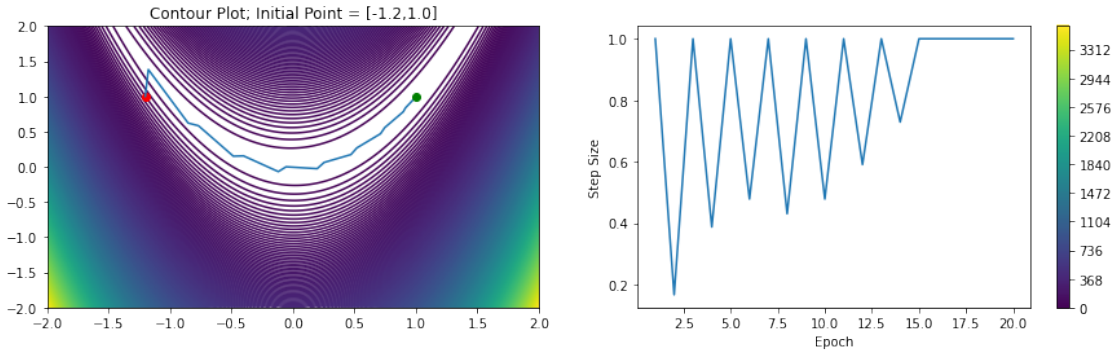
Steepest Descent Iterates and Step sizes

```
/var/folders/sm/_xvm618541nc11hb7mv_kpj40000gn/T/ipykernel_9281/1545082446.py:17
: MatplotlibDeprecationWarning: Starting from Matplotlib 3.6, colorbar() will
steal space from the mappable's axes, rather than from the current axes, to
place the colorbar.  To silence this warning, explicitly pass the 'ax' argument
```

to colorbar().
  fig.colorbar(cp)



Newtons Method Iterates and Step sizes



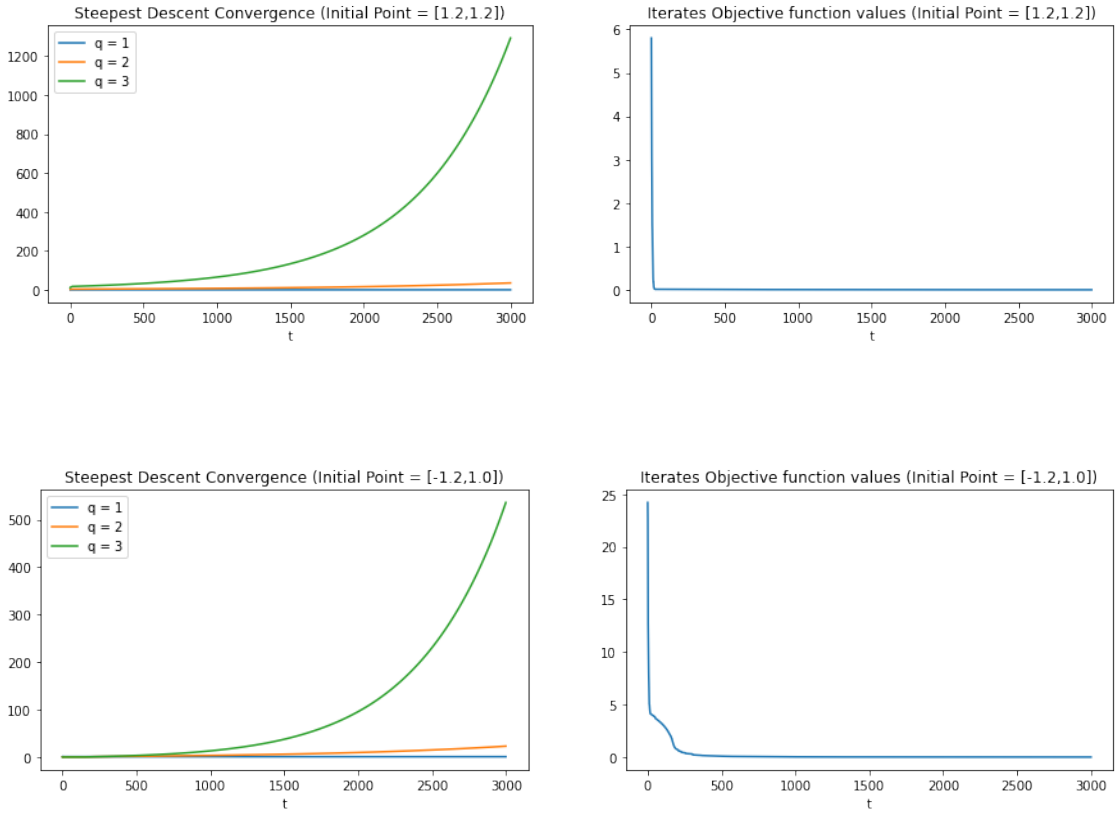
Rate Of Convergence Allowing the algorithms to run upto a maximum of 3000 iterations without any stopping criterion in between.

[31]:
```
def convergence(iterates,minima,q):

    convergence_values = []

    for i in range(1,len(iterates)):
        n1 = np.linalg.norm(iterates[i]-minima)
        n2 = np.linalg.norm(iterates[i-1]-minima)

        conv_value = n1 / np.power(n2,q)
        convergence_values.append(conv_value)
    return convergence_values
```

```
def plot_convergence_and_fnvalues(function_values,convergences,title):

    fig,ax = plt.subplots(1,2)



    c_indices = [j for j in range(len(convergences[0]))]
    f_indices = [j for j in range(len(function_values)) ]

    ax[0].set_title(title[0])

    fig.set_figwidth(15)
    for k in range(len(convergences)):
        ax[0].plot(c_indices,convergences[k],label='q = ' + str(k+1))

    ax[0].legend(loc = "upper left")

    ax[0].set_xlabel('t')
    ax[1].set_title(title[1])
    ax[1].plot(f_indices,function_values)
    ax[1].set_xlabel('t')

    plt.show()
```

[32]:
```
initial_point = np.array([[1.2],[1.2]])
iterations_steep_p1, iterates_steep_p1, function_values_steep_p1,␣
 ↪step_sizes_steep_p1 =␣
 ↪steepest_descent_rosenbrock(initial_point,no_stopping_criterion=True)
initial_point = np.array([[-1.2],[1.0]])
iterations_steep_p2, iterates_steep_p2, function_values_steep_p2,␣
 ↪step_sizes_steep_p2 =␣
 ↪steepest_descent_rosenbrock(initial_point,no_stopping_criterion=True)


convergences_p1 = []
convergences_p2 = []


for i in range(1,4):
    convergences_p1.append(convergence(iterates_steep_p1,np.array([[1.0],[1.
 ↪0]]),i))
    convergences_p2.append(convergence(iterates_steep_p2,np.array([[1.0],[1.
 ↪0]]),i))

plot_convergence_and_fnvalues(function_values_steep_p1,convergences_p1,['Steepest␣
 ↪Descent Convergence (Initial Point = [1.2,1.2])', 'Iterates Objective function␣
 ↪values (Initial Point = [1.2,1.2])'])
```

```
plot_convergence_and_fnvalues(function_values_steep_p2,convergences_p2,['Steepest␣
↪Descent Convergence (Initial Point = [-1.2,1.0])', 'Iterates Objective␣
↪function values (Initial Point = [-1.2,1.0])'])
```

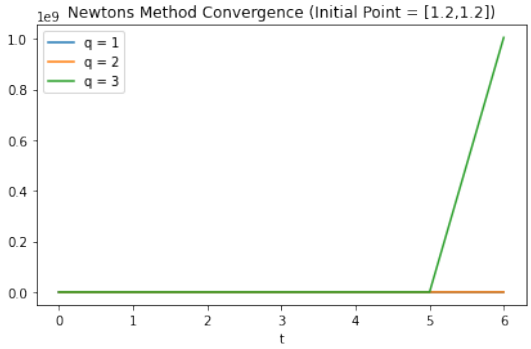


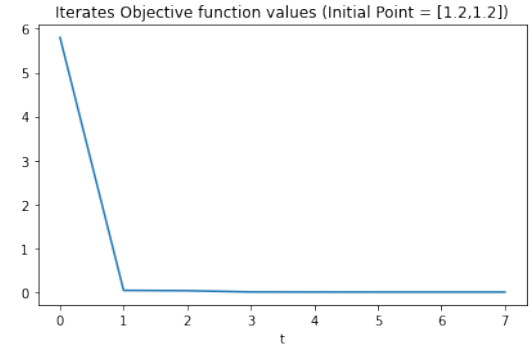
1.0.1 Rate of Convergence of Steepest Descent Method

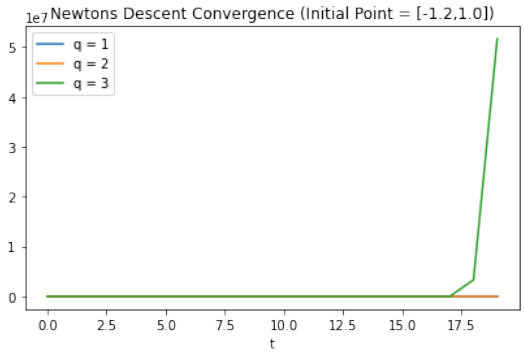
- We can see that the convergence value for q=1 is almost a constant that we can bound it by some positive constant M
- We can see a positive slope in the convergence function for $q = 2$ and $q = 3$ towards the end.
- The rate of convergence of the steepest descent method is SuperLinear for both the initial points.

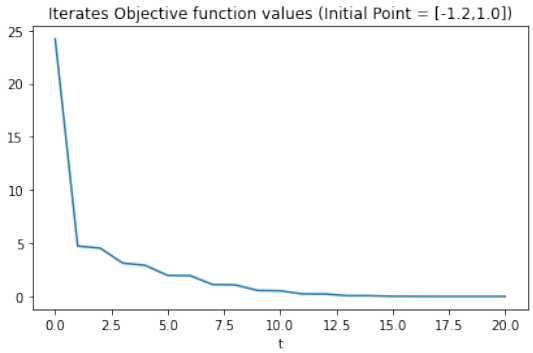
[33]:
```
initial_point = np.array([[1.2],[1.2]])
iterations_newton_p1, iterates_newton_p1, function_values_newton_p1,␣
↪step_sizes_newton_p1 =␣
↪newton_rosenbrock(initial_point,no_stopping_criterion=False)
initial_point = np.array([[-1.2],[1.0]])
iterations_newton_p2, iterates_newton_p2, function_values_newton_p2,␣
↪step_sizes_newton_p2 =␣
↪newton_rosenbrock(initial_point,no_stopping_criterion=False)
```

```
convergences_p1 = []
convergences_p2 = []


for i in range(1,4):

    convergences_p1.append(convergence(iterates_newton_p1,np.array([[1.0],[1.
 ↪0]]),i))
    convergences_p2.append(convergence(iterates_newton_p2,np.array([[1.0],[1.
 ↪0]]),i))

plot_convergence_and_fnvalues(function_values_newton_p1[:
 ↪20],convergences_p1,['Newtons Method Convergence (Initial Point = [1.2,1.2])',␣
 ↪'Iterates Objective function values (Initial Point = [1.2,1.2])'])
plot_convergence_and_fnvalues(function_values_newton_p2[:
 ↪150],convergences_p2,['Newtons Descent Convergence (Initial Point = [-1.2,1.
 ↪0])', 'Iterates Objective function values (Initial Point = [-1.2,1.0])'])
```









1.0.2 Rate of Convergence of Newtons Method

- We can see that the convergence value for both q=2 and q=1 is almost a constant that we can bound it by some positive constant M
- We can see a huge convergence value for $q = 3$ (1e7) that makes it difficult to find a bound for the convergence value as
  $k \to \infty$
- The rate of convergence of the steepest descent method is Quadratic for both the initial points. (A sequence that converges quadratic also converges superlinearly)

[34]:
```
def steepest_descent_rosenbrock_scipy(point):
    current_point = point.copy()
    previous_point = point.copy()
    iterations =0
    iterates = []
    while True:
        iterations+=1
        iterates.append(current_point)
        previous_point = current_point.copy()
        current_gradient = approx_fprime(current_point.squeeze(),rosenbrock)  #␣
↪Built in function for calculating the gradient
        current_gradient = np.expand_dims(current_gradient,axis=1)
        #current_gradient = np.expand_dims(rosenbrock_jacobian(current_point.
↪squeeze()),1)

        # Built in Function for calculating the step length which satisifies the␣
↪Wolfe Conditions
        step_length = line_search(rosenbrock,rosenbrock_jacobian,current_point.
↪squeeze(),-get_unit_vector(current_gradient).squeeze())
        if step_length[0] == None:
            break
        current_point += step_length[0]*(-get_unit_vector(current_gradient))
    return iterations,iterates[-1]


def newtons_method_scipy(point):
    current_point = point.copy()
    ans =␣
↪minimize(rosenbrock,current_point,method='Newton-CG',jac=rosenbrock_jacobian,hess=rosenbrock_
    return ans
```

[35]:
```
import datetime
def compare_steepest(point):
    init_point = point.copy()
    t1 = datetime.datetime.now()
    it1,f1 = steepest_descent_rosenbrock(init_point,return_extra_info=False)
    t2 = datetime.datetime.now()
    print("Time taken Steepest Descent (Own Implementation) =" + str(t2-t1))
```

```
    print("Iterations taken = " + str(it1))
    print("Final Value after convergence" + str(f1))

    print("-----------------------------------------------")

    t1 = datetime.datetime.now()
    it2,f2 = steepest_descent_rosenbrock_scipy(init_point)
    t2 = datetime.datetime.now()
    print("Time taken Steepest Descent (BuiltIn Implementation) =" + str(t2-t1))
    print("Iterations taken = " + str(it2))
    print("Final Value after convergence" + str(f2))

def compare_newton(point):

    init_point = point.copy()
    t1 = datetime.datetime.now()
    it1,f1 = newton_rosenbrock(init_point,return_extra_info=False)
    t2 = datetime.datetime.now()
    print("Time taken Newtons Method (Own Implementation) =" + str(t2-t1))
    print("Iterations taken = " + str(it1))
    print("Final Value after convergence" + str(f1))

    print("-----------------------------------------------")

    t1 = datetime.datetime.now()
    res =␣
 ↪minimize(rosenbrock,init_point,method='Newton-CG',jac=rosenbrock_jacobian,hess=rosenbrock_hes
    it2,f2,jac = res['nit'],res['x'] ,res['jac']
    t2 = datetime.datetime.now()
    print("Time taken Newtons Method (BuiltIn Implementation) =" + str(t2-t1))
    print("Iterations taken = " + str(it2))
    print("Final Value after convergence" + str(f2))
    print("Jacobian" + str(jac))
```

1.0.3 Initial Point (1.2,1.2)

[36]:
```
import warnings
warnings.filterwarnings("ignore")

compare_steepest(np.array([[1.2],[1.2]]))
```

```
Time taken Steepest Descent (Own Implementation) =0:00:02.573151
Iterations taken = 8332
Final Value after convergence[[1.00018487]
 [1.00036221]]
-----------------------------------------------
Time taken Steepest Descent (BuiltIn Implementation) =0:00:00.001228
```

```
Iterations taken = 8
Final Value after convergence[[1.0306002 ]
 [1.06217019]]
```

[37]:
```
compare_newton(np.array([[1.2],[1.2]]))
```

```
Time taken Newtons Method (Own Implementation) =0:00:00.000844
Iterations taken = 7
Final Value after convergence[[1.]
 [1.]]
-----------------------------------------------
Time taken Newtons Method (BuiltIn Implementation) =0:00:00.001251
Iterations taken = 12
Final Value after convergence[0.99999998 0.99999997]
Jacobian[ 4.62401128e-06 -2.32450401e-06]
```

Initial Point (-1.2,1)

[38]:
```
compare_steepest(np.array([[-1.2],[1.0]]))
```

```
Time taken Steepest Descent (Own Implementation) =0:00:02.653251
Iterations taken = 8843
Final Value after convergence[[0.99982214]
 [0.99965198]]
-----------------------------------------------
Time taken Steepest Descent (BuiltIn Implementation) =0:00:00.018267
Iterations taken = 137
Final Value after convergence[[1.01904608]
 [1.03840753]]
```

[39]:
```
compare_newton(np.array([[-1.2],[1.0]]))
```

```
Time taken Newtons Method (Own Implementation) =0:00:00.001793
Iterations taken = 20
Final Value after convergence[[1.]
 [1.]]
-----------------------------------------------
Time taken Newtons Method (BuiltIn Implementation) =0:00:00.005532
Iterations taken = 83
Final Value after convergence[0.9999826  0.99996514]
Jacobian[ 0.00510412 -0.00256439]
```

Tabular Summary

Steepest Descent

Initial Point	Iterations(Steepest Descent- Own)	Time(seconds)(Steepest Descent -Own)	Minima(Steepest Descent - Own)	Iterations(Steepest Descent- Scipy)	Time (Steep- est Descent -Scipy)	Minima(Steepest Descent - Scipy)
(1.2,1.2)	8332	2.603 s	(1.00018,1.00036)	87	0.001796 s	(1.030602,1.06217019)
(-1.2,1.0)	8843	2.642 s	(0.9982,0.9937)	137	0.0195415 s	(1.01904608,1.03840753)

Newtons Method

Initial Point	Iterations(Newtons Method - Own)	Time (New- tons Method -Own)	Minima(Newtons Method - Own)	Iterations(Newtons Method - Scipy)	Time (New- tons Method -Scipy)	Minima(Newtons Method - Scipy)
(1.2,1.2)	7	0.001403 s	(1.0,1.0)	12	0.001637 s	$(0.99999998 , 0.99999997) $
(-1.2,1.0)	20	0.001973 s	(1.0,1.0)	83	0.007742 s	(0.9999826, 0.99996514)

[40]:
```
def calculate_jacobian(iterates ,init_point = '[1.2,1.2]'):
    jacobian_values = []

    for iterate in iterates:

        jacobian_values.append(rosenbrock_jacobian(iterate.squeeze()))

    j_df = pd.DataFrame({'Jacobian' + init_point : jacobian_values})

    return j_df

df1 = calculate_jacobian(iterates_newton_p1)
df2 = calculate_jacobian(iterates_newton_p2)
```

[41]:
```
print("Jacobian at every point Starting point = [1.2,1.2]")
print(df1)
print("")
print("Final Jacobian computed by BuiltIn Function; Starting point = [1.2,1.2] =␣
↪[4.62401128e-06 -2.32450401e-06] ")
```

```
Jacobian at every point Starting point = [1.2,1.2]
                                  Jacobian[1.2,1.2]
```

```
0                                    [115.6, -48.0]
1         [0.3998062031977625, -0.003331945022955196]
2            [7.148607040471751, -3.2838451163524685]
3         [0.20936514679767954, -0.05010289805444934]
4          [0.9873185145379902, -0.48996432950496605]
5     [0.0027399568453320496, -0.0005540365485501297]
6    [0.0002666183459213766, -0.00013285755184888615]
7   [2.0159429681633214e-10, -4.0811798385220754e-11]

Final Jacobian computed by BuiltIn Function; Starting point = [1.2,1.2] =
[4.62401128e-06 -2.32450401e-06]
```

[42]:
```
print("Jacobian at every point Starting point = [-1.2,1.0]")
print(df2)
print("")
print("Final Jacobian computed by BuiltIn Function; Starting point = [-1.2,1.0]␣
 ↪= [ 0.00510412 -0.00256439] ")
```

```
Jacobian at every point Starting point = [-1.2,1.0]
                                   Jacobian[1.2,1.2]
0                        [-215.6, -87.99999999999999]
1          [-4.637816414622754, -0.12220679207182172]
2           [-39.492743756125435, -21.00249687429292]
3            [-5.711747524412797, -1.417011050189254]
4          [-19.370492435430993, -16.927198128037226]
5           [-3.904992424912343, -1.3713548761476402]
6          [-6.2111711249800585, -16.700786026664126]
7           [-2.200575564508425, -0.7992260522746861]
8           [3.5964685796050166, -13.215228647819421]
9          [-1.1698134554878552, -0.6384371793719257]
10              [8.2131542645363, -9.834813218925264]
11        [-0.4644390411818759, -0.47619638896049254]
12           [10.197438287205662, -7.564797769599063]
13        [-0.17152560709910158, -0.22603483873173 68]
14            [8.107084701620765, -4.634447717575774]
15     [-0.047032802743123464, -0.06696307053264405]
16          [2.5026595993644074, -1.2633758122297056]
17    [-0.003736353326803156, -0.001105833993975125]
18     [0.0035157150635023835, -0.001761144811429638]
19  [-7.24595275148132e-09, -2.1460611066004276e-09]
20   [4.440892098500626e-14, -2.220446049250313e-14]

Final Jacobian computed by BuiltIn Function; Starting point = [-1.2,1.0] = [
0.00510412 -0.00256439]
```

Comparisons

- We can see differences in both the run time and results of builtin functions and custom made functions

- The minima computed by the builtin and custom implementations are almost similar.
- Steepest Descent done through builtin functions calculates the best step length to take better than the custom implementation. This explains the low run time and number of iterations taken to converge.
- The Newtons implementation on the other hand is slightly faster on using the custom implementation rather than builtin library functions.
- The builtin Newtons method does more iterations than the custom implementation which might be due to less amounts of update performed by the builtin function at each iteration.