LAB 1 HOMEWORK 3

Gökce Sucu 246112

EXERCISE 1: GRADIENT DESCENT ON RF

In this pary we will use Rosenbrock function: $f(x, y) = (1 - x)^2 + 100(y - x^2)^2$

1. 3D PLOTTING

1. Importing The Libraries

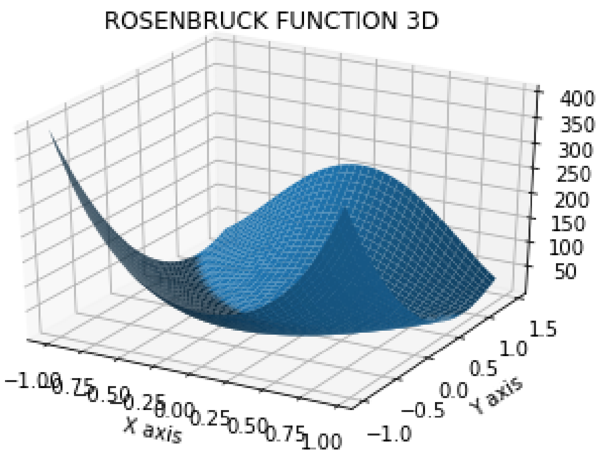
In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

2. Defining The Rosenbrock Function

In [2]:
```
def rosenbrock_function(x,y,a,b):
    return (a-x)**2+b*(y-x**2)**2
```

3. Plotting

In [3]:
```
rosenbrock_plot = plt.figure()
axis = rosenbrock_plot.gca(projection='3d')

# tuples
X = np.arange(-1, 1, 0.05)
Y = np.arange(-1, 1.5, 0.05)


X_grid = np.meshgrid(X,Y)[0]
Y_grid = np.meshgrid(X,Y)[1]


#a=1 and b=100
Z =rosenbrock_function(X_grid,Y_grid,1,100)

#texts on plots
plt.xlabel('X axis')
plt.ylabel('Y axis')


# Plot the surface
surf = axis.plot_surface(X_grid, Y_grid, Z)
plt.title('ROSENBRUCK FUNCTION 3D')
plt.show()
```



2. PARTIAL DERIVATIVES

$$\nabla f(x, y, a, b) = \begin{bmatrix} f_x \\ f_y \end{bmatrix} = \begin{bmatrix} -2a + 2x - 4bxy + 4bx^3 \\ 2by - 2bx^2 \end{bmatrix}$$

$$\nabla f(x, y, 1, 100) = \begin{bmatrix} -2 + 2x - 400xy + 400x^3 \\ 200y - 200x^2 \end{bmatrix}$$

In [4]:
```
#defining the partial derivatives
def partial_f_x(x,y,a,b):
    return -2*a+2*x-4*b*x*y+4*b**3


def partial_f_y(x,y,a,b):
    return 2*b*y-2*b*x**2
```

3. GRADIENT OF FUNCTION

In [5]:
```python
#defining the rosenbruck function
def rosenbrock_function(x,y,a,b):
    return (a-x)**2+b*(y-x**2)**2

#defining the gradient of rosenbruck function
def gradient_rosenbruck(x,y,a,b):
    return np.array([4*b*(x**3)-4*b*x*y+2*x-2*a,2*b*y-2*b*(x**2)])
```

4. OPTIMIZATION BY GD

4.1 Creating a GD Algorithm

First, we should define gradient descent alogrithm.

In [6]:
```python
#defining a gradient descent function especially for rosenbruck function
def gradient_descent(x,y,a,b,alpha,k):
    xy_old = np.array([x,y])
    approach = []
    minimum_point =np.array([a,a**2])
    for i in range(0,k):
        xy_next = xy_old- alpha*gradient_rosenbruck(xy_old[0],xy_old[1],a,b)
        approach.append(xy_next)
        xy_old=xy_next
    print(f'error{minimum_point-approach[-1]}\n',approach[-1])
```

4.2 Finding The Best Hyperparameters

In [7]:
```python
#try 1: x=2, y=2, a=1, b=100, alpha=0.1, k = 5
gradient_descent(2,2,1,100,0.1,5)
```

```
error[ 1.57913917e+242 -4.99578346e+161]
 [-1.57913917e+242  4.99578346e+161]
```

In [8]:
```python
#try 2 : x=2, y=2, a=1, b=100, alpha=0.001, k = 10
gradient_descent(2,2,1,100,0.001,10)
```

```
error[-0.27645524 -0.62838782]
 [1.27645524 1.62838782]
```

In [9]:
```python
#try 3 x=2, y=2, a=1, b=100, alpha=0.0001, k = 100
gradient_descent(2,2,1,100,0.0001,100)
```

```
error[-0.46542458 -1.14889175]
 [1.46542458 2.14889175]
```

4.3 Best Performance

Best performance is for starting point at (2,2) when step size 0.0001 and 300000 iteration. Error is so small.

In [10]:
```python
#x=2, y=2, a=1, b=100, alpha=0.0001, k = 300000
gradient_descent(2,2,1,100,0.0001,300000)
```

```
error[-6.70883471e-06 -1.34445604e-05]
 [1.00000671 1.00001344]
```

5. VISUALIZE THE TRAJECTORY

5. VISUALIZE THE TRAJECTORY

5.1 Creating Path Of GD

To be able to plot the GD way, we have to define a new function which gives the each iteration of GD.

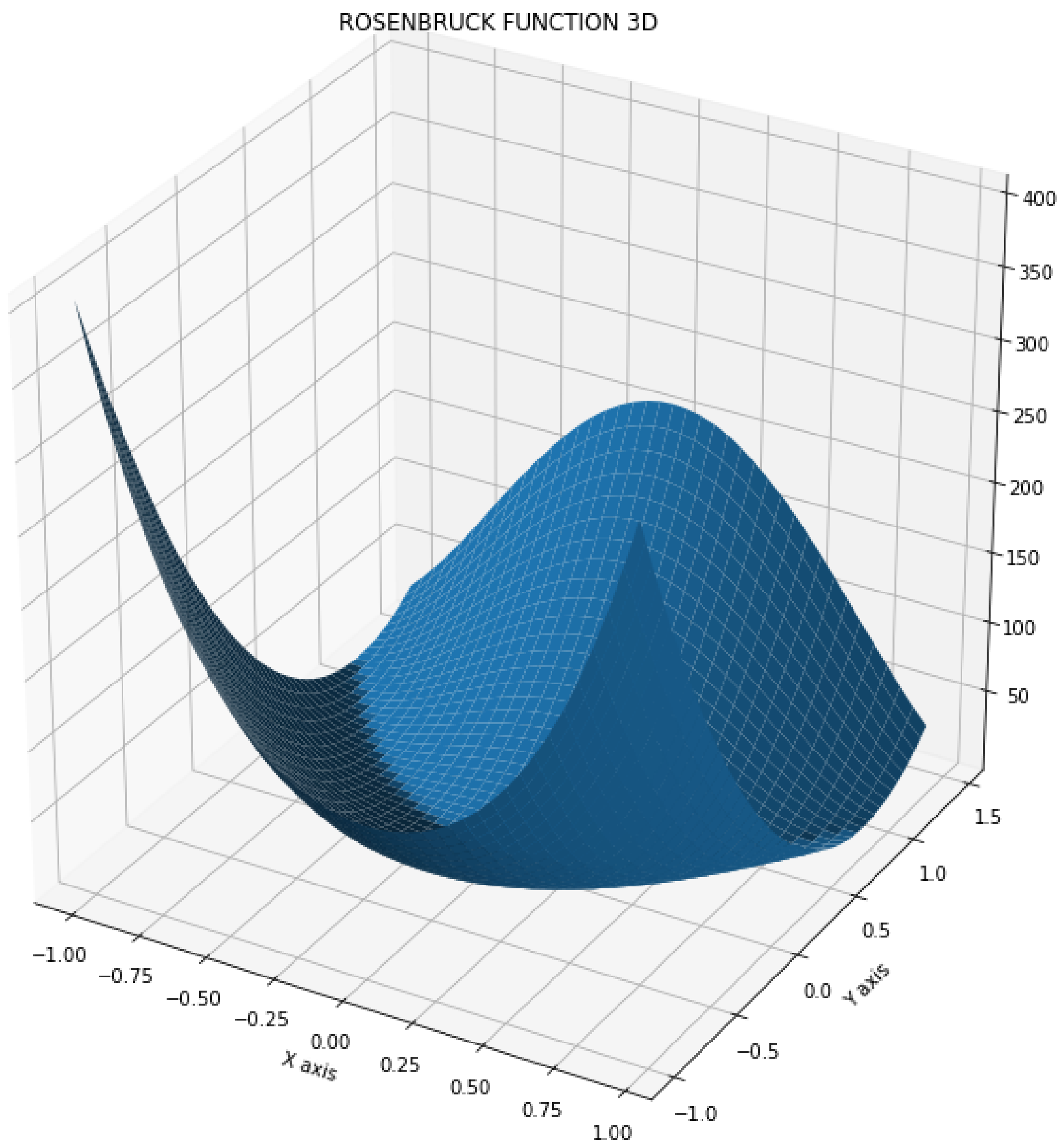
In [11]:
```
def path_of_grad(x,y,a,b,alpha,k):
    xy_old = np.array([x,y])
    path = []
    for i in range(0,k):
        xy_next = xy_old- alpha*gradient_rosenbruck(xy_old[0],xy_old[1],a,b)
        path.append(xy_next)
        xy_old=xy_next
    return path
```

5.2 Choosing The Hyperparameters

In [12]:
```
#x=10, y=10, a=1, b=100, alpha=0.01, k = 10
path_of_grad(10,10,1,100,0.0002,5)
```

Out[12]:
```
[array([-62.0036,  13.6   ]),
 array([18940.12309349,   166.83385652]),
 array([-5.43548325e+11,  1.43492907e+07]),
 array([1.28470813e+34, 1.18177912e+22]),
 array([-1.69630289e+101,  6.60189988e+066])]
```

In [13]:
```
rosenbrock_plot = plt.figure(figsize=(12,12))
axis = rosenbrock_plot.gca(projection='3d')

# tuples
X = np.arange(-1, 1, 0.05)
Y = np.arange(-1, 1.5, 0.05)


X_grid = np.meshgrid(X,Y)[0]
Y_grid = np.meshgrid(X,Y)[1]


#a=1 and b=100
Z =rosenbrock_function(X_grid,Y_grid,1,100)

#texts on plots
plt.xlabel('X axis')
plt.ylabel('Y axis')


# Plot the surface
surf = axis.plot_surface(X_grid, Y_grid, Z)
plt.title('ROSENBRUCK FUNCTION 3D')


plt.show()
```



EXERCISE 2:LR WITH GRADIENT DESCENT

PART A: DATASETS

1. AIRFARE ANC DEMAND DATASET

In [14]:
```
#necessary libraries
import pandas as pd
```

A.1.1 Airfare and Demand: Reading The Dataset

First, we have to create headlines for this dataset. We would do this by using other dataset.

In [15]:
```
#creating the headlines
airport_dataset = pd.read_csv("airq402.data", sep='\s+',
names=["City 1",
"City 2",
"Average Fare 1",
"Distance",
"Average Weekly Passengers",
"Market Leading Airline",
"Market Share 1",
"Average  2",
"Low Price Airline",
"Market Share 2",
"Price"], na_values='?')
```

In [16]:
```
airport_dataset
```

Out[16]:

	City 1	City 2	Average Fare 1	Distance	Average Weekly Passengers	Market Leading Airline	Market Share 1	Average 2	Low Price Airline	Market Share 2	Price
0	CAK	ATL	114.47	528	424.56	FL	70.19	111.03	FL	70.19	111.03
1	CAK	MCO	122.47	860	276.84	FL	75.10	123.09	DL	17.23	118.94
2	ALB	ATL	214.42	852	215.76	DL	78.89	223.98	CO	2.77	167.12
3	ALB	BWI	69.40	288	606.84	WN	96.97	68.86	WN	96.97	68.86
4	ALB	ORD	158.13	723	313.04	UA	39.79	161.36	WN	15.34	145.42
...	...	...	...	...	...	...	...	...	...	...	...
995	SYR	TPA	136.16	1104	184.34	US	33.37	135.82	DL	28.65	118.51
996	TLH	TPA	83.28	200	232.71	FL	99.57	82.55	FL	99.57	82.55
997	TPA	IAD	159.97	814	843.80	US	46.19	159.65	DL	13.89	159.02
998	TPA	PBI	73.57	174	214.45	WN	99.74	73.44	WN	99.74	73.44
999	IAD	PBI	126.67	859	475.65	US	56.28	129.92	DL	38.57	121.94

1000 rows × 11 columns

A.1.2 Airfare and Demand: Converting Any Non-Numeric Values

In [17]:
```
#converting object values to numeric
airport_dataset_converted = pd.get_dummies(airport_dataset)

#displaying the new data which is converted to numerical values
airport_dataset_converted
```

Out[17]:

	Average Fare 1	Distance	Average Weekly Passengers	Market Share 1	Average 2	Market Share 2	Price	City 1_ABQ	City 1_ACY	City 1_ALB	...	Low Pric Airline_G
0	114.47	528	424.56	70.19	111.03	70.19	111.03	0	0	0	...	
1	122.47	860	276.84	75.10	123.09	17.23	118.94	0	0	0	...	
2	214.42	852	215.76	78.89	223.98	2.77	167.12	0	0	1	...	
3	69.40	288	606.84	96.97	68.86	96.97	68.86	0	0	1	...	
4	158.13	723	313.04	39.79	161.36	15.34	145.42	0	0	1	...	
...	...	...	...	...	...	...	...	...	...	...	...	
995	136.16	1104	184.34	33.37	135.82	28.65	118.51	0	0	0	...	
996	83.28	200	232.71	99.57	82.55	99.57	82.55	0	0	0	...	
997	159.97	814	843.80	46.19	159.65	13.89	159.02	0	0	0	...	
998	73.57	174	214.45	99.74	73.44	99.74	73.44	0	0	0	...	
999	126.67	859	475.65	56.28	129.92	38.57	121.94	0	0	0	...	

1000 rows × 217 columns

Now, in our table there is no any text, there are just numerical values.

A.1.3 Airfare and Demand: Dropping NaN Values

In [18]:
```
#checking if there is any column that consists NaN values.
airport_dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 11 columns):
 #   Column                     Non-Null Count  Dtype
---  ------                     --------------  -----
 0   City 1                     1000 non-null   object
 1   City 2                     1000 non-null   object
 2   Average Fare 1             1000 non-null   float64
 3   Distance                   1000 non-null   int64
 4   Average Weekly Passengers  1000 non-null   float64
 5   Market Leading Airline     1000 non-null   object
 6   Market Share 1             1000 non-null   float64
 7   Average  2                 1000 non-null   float64
 8   Low Price Airline          1000 non-null   object
 9   Market Share 2             1000 non-null   float64
 10  Price                      1000 non-null   float64
dtypes: float64(6), int64(1), object(4)
memory usage: 86.1+ KB
```

So, there is no any NaN value.

A.1.4 Airfare and Demand : Splitting Into Train and Test Sets

In [19]:
```
#dataset informatio
n=airport_dataset_converted.shape[0]
k= int(n*80/100)

#train data %80
airfare_demand_train = airport_dataset_converted.iloc[:k,:]

#test data %20
airfare_demand_test = airport_dataset_converted.iloc[k:,:]
```

In [20]:
```
airfare_demand_train
```

Out[20]:

	Average Fare 1	Distance	Average Weekly Passengers	Market Share 1	Average 2	Market Share 2	Price	City 1_ABQ	City 1_ACY	City 1_ALB	...	Low Pric Airline_G
0	114.47	528	424.56	70.19	111.03	70.19	111.03	0	0	0	...	
1	122.47	860	276.84	75.10	123.09	17.23	118.94	0	0	0	...	
2	214.42	852	215.76	78.89	223.98	2.77	167.12	0	0	1	...	
3	69.40	288	606.84	96.97	68.86	96.97	68.86	0	0	1	...	
4	158.13	723	313.04	39.79	161.36	15.34	145.42	0	0	1	...	
...	...	...	...	...	...	...	...	...	...	...	...	
795	124.05	616	568.26	66.67	121.22	66.67	121.22	0	0	0	...	
796	242.28	675	215.10	78.06	241.33	1.86	145.18	0	0	0	...	
797	172.95	1448	317.82	78.45	170.44	7.31	164.12	0	0	0	...	
798	133.87	907	227.06	75.34	133.59	10.14	130.38	0	0	0	...	
799	94.97	443	471.19	88.37	92.06	88.37	92.06	0	0	0	...	

800 rows × 217 columns

In [21]:
```
airfare_demand_test
```

Out[21]:

	Average Fare 1	Distance	Average Weekly Passengers	Market Share 1	Average 2	Market Share 2	Price	City 1_ABQ	City 1_ACY	City 1_ALB	...	Low Pric Airline_G
800	133.44	822	196.73	66.40	134.34	11.93	125.38	0	0	0	...	
801	162.00	1751	229.34	65.02	158.65	11.18	151.64	0	0	0	...	
802	166.19	1977	231.63	42.13	159.59	17.92	151.51	0	0	0	...	
803	128.97	612	442.17	83.70	125.31	83.70	125.31	0	0	0	...	
804	164.99	1183	1473.80	32.87	204.45	17.54	128.88	0	0	0	...	
...	...	...	...	...	...	...	...	...	...	...	...	
995	136.16	1104	184.34	33.37	135.82	28.65	118.51	0	0	0	...	
996	83.28	200	232.71	99.57	82.55	99.57	82.55	0	0	0	...	
997	159.97	814	843.80	46.19	159.65	13.89	159.02	0	0	0	...	
998	73.57	174	214.45	99.74	73.44	99.74	73.44	0	0	0	...	
999	126.67	859	475.65	56.28	129.92	38.57	121.94	0	0	0	...	

200 rows × 217 columns

A.1.5 Airfare and Demand: Dimensions of Train and Test Sets

In [22]:
```
print(airfare_demand_train.shape)
print(airfare_demand_test.shape)
```

```
(800, 217)
(200, 217)
```

A.2 WINE QUALITY

A.2.1 Wine Quality: Reading The Dataset

In [23]:
```
wine_quality_dataset = pd.read_csv('winequality-red.csv', engine='python', sep=";", decir
```

In [24]:
```
wine_quality_dataset
```

Out[24]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	9.4	5
1	7.8	0.880	0.00	2.6	0.098	25.0	67.0	0.99680	3.20	0.68	9.8	5
2	7.8	0.760	0.04	2.3	0.092	15.0	54.0	0.99700	3.26	0.65	9.8	5
3	11.2	0.280	0.56	1.9	0.075	17.0	60.0	0.99800	3.16	0.58	9.8	6
4	7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	9.4	5
...	...	...	...	...	...	...	...	...	...	...	...	...
1594	6.2	0.600	0.08	2.0	0.090	32.0	44.0	0.99490	3.45	0.58	10.5	5
1595	5.9	0.550	0.10	2.2	0.062	39.0	51.0	0.99512	3.52	0.76	11.2	6
1596	6.3	0.510	0.13	2.3	0.076	29.0	40.0	0.99574	3.42	0.75	11.0	6
1597	5.9	0.645	0.12	2.0	0.075	32.0	44.0	0.99547	3.57	0.71	10.2	5
1598	6.0	0.310	0.47	3.6	0.067	18.0	42.0	0.99549	3.39	0.66	11.0	6

1599 rows × 12 columns

A. 2.2 Wine Quality: Converting Any Non-Numerical Values

In [25]:
```
wine_quality_dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1599 entries, 0 to 1598
Data columns (total 12 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   fixed acidity         1599 non-null   float64
 1   volatile acidity      1599 non-null   float64
 2   citric acid           1599 non-null   float64
 3   residual sugar        1599 non-null   float64
 4   chlorides             1599 non-null   float64
 5   free sulfur dioxide   1599 non-null   float64
 6   total sulfur dioxide  1599 non-null   float64
 7   density               1599 non-null   float64
 8   pH                    1599 non-null   float64
 9   sulphates             1599 non-null   float64
 10  alcohol               1599 non-null   float64
 11  quality               1599 non-null   int64
dtypes: float64(11), int64(1)
memory usage: 150.0 KB
```

According to information, every data is numerical. So, no need to convert.

A 2.3 Wine Quality: Dropping NaN Values

In [26]:
```
wine_quality_dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1599 entries, 0 to 1598
Data columns (total 12 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   fixed acidity         1599 non-null   float64
 1   volatile acidity      1599 non-null   float64
 2   citric acid           1599 non-null   float64
 3   residual sugar        1599 non-null   float64
 4   chlorides             1599 non-null   float64
 5   free sulfur dioxide   1599 non-null   float64
 6   total sulfur dioxide  1599 non-null   float64
 7   density               1599 non-null   float64
 8   pH                    1599 non-null   float64
 9   sulphates             1599 non-null   float64
 10  alcohol               1599 non-null   float64
 11  quality               1599 non-null   int64
dtypes: float64(11), int64(1)
memory usage: 150.0 KB
```

According to the information,there is no any NaN values. So no need to drop.

A .2.4 Wine Quality: Splitting Into Train and Test Sets

In [27]:
```
#need to translate non integer numbers to integers by math.ceil()
import math
```

In [28]:
```
nu = wine_quality_dataset.shape[0]
ku = math.ceil(nu*80/100)

#train sets %80
wine_quality_train = wine_quality_dataset.iloc[:ku,:]

#test sets %20
wine_quality_test= wine_quality_dataset.iloc[ku:,:]
```

In [29]:
```
wine_quality_train
```

Out[29]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	9.4	5
1	7.8	0.880	0.00	2.6	0.098	25.0	67.0	0.99680	3.20	0.68	9.8	5
2	7.8	0.760	0.04	2.3	0.092	15.0	54.0	0.99700	3.26	0.65	9.8	5
3	11.2	0.280	0.56	1.9	0.075	17.0	60.0	0.99800	3.16	0.58	9.8	6
4	7.4	0.700	0.00	1.9	0.076	11.0	34.0	0.99780	3.51	0.56	9.4	5
...	...	...	...	...	...	...	...	...	...	...	...	...
1275	8.0	0.715	0.22	2.3	0.075	13.0	81.0	0.99688	3.24	0.54	9.5	6
1276	8.5	0.400	0.40	6.3	0.050	3.0	10.0	0.99566	3.28	0.56	12.0	4
1277	7.0	0.690	0.00	1.9	0.114	3.0	10.0	0.99636	3.35	0.60	9.7	6
1278	8.0	0.715	0.22	2.3	0.075	13.0	81.0	0.99688	3.24	0.54	9.5	6
1279	9.8	0.300	0.39	1.7	0.062	3.0	9.0	0.99480	3.14	0.57	11.5	7

1280 rows × 12 columns

In [30]:
```
wine_quality_test
```

Out[30]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
1280	7.1	0.460	0.20	1.9	0.077	28.0	54.0	0.99560	3.37	0.64	10.4	6
1281	7.1	0.460	0.20	1.9	0.077	28.0	54.0	0.99560	3.37	0.64	10.4	6
1282	7.9	0.765	0.00	2.0	0.084	9.0	22.0	0.99619	3.33	0.68	10.9	6
1283	8.7	0.630	0.28	2.7	0.096	17.0	69.0	0.99734	3.26	0.63	10.2	6
1284	7.0	0.420	0.19	2.3	0.071	18.0	36.0	0.99476	3.39	0.56	10.9	5
...	...	...	...	...	...	...	...	...	...	...	...	...
1594	6.2	0.600	0.08	2.0	0.090	32.0	44.0	0.99490	3.45	0.58	10.5	5
1595	5.9	0.550	0.10	2.2	0.062	39.0	51.0	0.99512	3.52	0.76	11.2	6
1596	6.3	0.510	0.13	2.3	0.076	29.0	40.0	0.99574	3.42	0.75	11.0	6
1597	5.9	0.645	0.12	2.0	0.075	32.0	44.0	0.99547	3.57	0.71	10.2	5
1598	6.0	0.310	0.47	3.6	0.067	18.0	42.0	0.99549	3.39	0.66	11.0	6

319 rows × 12 columns

A. 2. 5. Wine Quality: Dimensions of Train and Test

```
In [31]: print(wine_quality_train.shape)
         print(wine_quality_test.shape)
```

```
(1280, 12)
(319, 12)
```

A. 3. PARKINSONS DATASET

A.3.1 Parkinson: Reading The Dataset

```
In [32]: parkinson_dataset = pd.read_csv("airq402.data", sep='\s+',
         names=["Age",
         "Sex",
         "Test Time",
         "Motor_UPDRS",
         "Total_UPDRS",
          "Jitter%"  ,
         "Jitter(Abs)",
         "Jitter:RAP",
         "Jitter:PPq5",
         "Jitter:DDP",
         "Shimmer"
         ], na_values='?')
```

```
In [33]: parkinson_dataset
```

Out[33]:

	Age	Sex	Test Time	Motor_UPDRS	Total_UPDRS	Jitter%	Jitter(Abs)	Jitter:RAP	Jitter:PPq5	Jitter:DDP
0	CAK	ATL	114.47	528	424.56	FL	70.19	111.03	FL	70.19
1	CAK	MCO	122.47	860	276.84	FL	75.10	123.09	DL	17.23
2	ALB	ATL	214.42	852	215.76	DL	78.89	223.98	CO	2.77
3	ALB	BWI	69.40	288	606.84	WN	96.97	68.86	WN	96.97
4	ALB	ORD	158.13	723	313.04	UA	39.79	161.36	WN	15.34
...	...	...	...	...	...	...	...	...	...	...
995	SYR	TPA	136.16	1104	184.34	US	33.37	135.82	DL	28.65
996	TLH	TPA	83.28	200	232.71	FL	99.57	82.55	FL	99.57
997	TPA	IAD	159.97	814	843.80	US	46.19	159.65	DL	13.89
998	TPA	PBI	73.57	174	214.45	WN	99.74	73.44	WN	99.74
999	IAD	PBI	126.67	859	475.65	US	56.28	129.92	DL	38.57

1000 rows × 11 columns

A.3.2 Parkinson: Converting Any Non Numerical Values

```
In [34]: parkinson_dataset_converted= pd.get_dummies(parkinson_dataset)
         parkinson_dataset_converted
```

Out[34]:

	Test Time	Motor_UPDRS	Total_UPDRS	Jitter(Abs)	Jitter:RAP	Jitter:DDP	Shimmer	Age_ABQ	Age_ACY
0	114.47	528	424.56	70.19	111.03	70.19	111.03	0	0
1	122.47	860	276.84	75.10	123.09	17.23	118.94	0	0
2	214.42	852	215.76	78.89	223.98	2.77	167.12	0	0
3	69.40	288	606.84	96.97	68.86	96.97	68.86	0	0
4	158.13	723	313.04	39.79	161.36	15.34	145.42	0	0
...	...	...	...	...	...	...	...	...	...
995	136.16	1104	184.34	33.37	135.82	28.65	118.51	0	0
996	83.28	200	232.71	99.57	82.55	99.57	82.55	0	0
997	159.97	814	843.80	46.19	159.65	13.89	159.02	0	0
998	73.57	174	214.45	99.74	73.44	99.74	73.44	0	0
999	126.67	859	475.65	56.28	129.92	38.57	121.94	0	0

1000 rows × 217 columns

A.3.3 Parkinson: Dropping The NaN Values

```
In [35]: parkinson_dataset.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1000 entries, 0 to 999
Data columns (total 11 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Age          1000 non-null   object
 1   Sex          1000 non-null   object
 2   Test Time    1000 non-null   float64
 3   Motor_UPDRS  1000 non-null   int64
 4   Total_UPDRS  1000 non-null   float64
 5   Jitter%      1000 non-null   object
 6   Jitter(Abs)  1000 non-null   float64
 7   Jitter:RAP   1000 non-null   float64
 8   Jitter:PPq5  1000 non-null   object
 9   Jitter:DDP   1000 non-null   float64
 10  Shimmer      1000 non-null   float64
dtypes: float64(6), int64(1), object(4)
memory usage: 86.1+ KB
```

According to the info, there is no any NaN values, so no need to drop.

A.3.4 Parkinson: Splitting Into Train and Test Sets

In [36]:
```
#number of data in the dataset
num = parkinson_dataset_converted.shape[0]
kum = math.ceil(num*80/100)

#train set  %80
parkinson_train= parkinson_dataset_converted.iloc[:kum,:]

#test set 520
parkinson_test= parkinson_dataset_converted.iloc[kum:,:]
```

In [37]:
```
parkinson_train
```

Out[37]:

	Test Time	Motor_UPDRS	Total_UPDRS	Jitter(Abs)	Jitter:RAP	Jitter:DDP	Shimmer	Age_ABQ	Age_ACY
0	114.47	528	424.56	70.19	111.03	70.19	111.03	0	0
1	122.47	860	276.84	75.10	123.09	17.23	118.94	0	0
2	214.42	852	215.76	78.89	223.98	2.77	167.12	0	0
3	69.40	288	606.84	96.97	68.86	96.97	68.86	0	0
4	158.13	723	313.04	39.79	161.36	15.34	145.42	0	0
...	...	...	...	...	...	...	...	...	...
795	124.05	616	568.26	66.67	121.22	66.67	121.22	0	0
796	242.28	675	215.10	78.06	241.33	1.86	145.18	0	0
797	172.95	1448	317.82	78.45	170.44	7.31	164.12	0	0
798	133.87	907	227.06	75.34	133.59	10.14	130.38	0	0
799	94.97	443	471.19	88.37	92.06	88.37	92.06	0	0

800 rows × 217 columns

In [38]:
```
parkinson_test
```

Out[38]:

	Test Time	Motor_UPDRS	Total_UPDRS	Jitter(Abs)	Jitter:RAP	Jitter:DDP	Shimmer	Age_ABQ	Age_ACY
800	133.44	822	196.73	66.40	134.34	11.93	125.38	0	0
801	162.00	1751	229.34	65.02	158.65	11.18	151.64	0	0
802	166.19	1977	231.63	42.13	159.59	17.92	151.51	0	0
803	128.97	612	442.17	83.70	125.31	83.70	125.31	0	0
804	164.99	1183	1473.80	32.87	204.45	17.54	128.88	0	0
...	...	...	...	...	...	...	...	...	...
995	136.16	1104	184.34	33.37	135.82	28.65	118.51	0	0
996	83.28	200	232.71	99.57	82.55	99.57	82.55	0	0
997	159.97	814	843.80	46.19	159.65	13.89	159.02	0	0
998	73.57	174	214.45	99.74	73.44	99.74	73.44	0	0
999	126.67	859	475.65	56.28	129.92	38.57	121.94	0	0

200 rows × 217 columns

A.3.5 Parkinson: Dimension of Train and Test Sets

In [39]:
```
print(parkinson_train.shape)
print(parkinson_test.shape)
```

```
(800, 217)
(200, 217)
```

PART B: LINEAR REGRESSION WITH REAL DATA

B.1. CREATING ALL FUNCTIONS

First, we will create all necessary functions.

1. Linear Regression Model

In [40]:
```
def lineer_regression_model(X,Y,beta):
    return np.matmul(X,beta)
```

2. Least Square Function

Now, we will define Least square function. $f(X, Y, B) = ||Y - XB|| = (Y - XB)^T(Y - XB)$

In [41]:
```
def least_square(X,Y,B):
    rss = ((Y-(X@B)).T)@(Y-(X@B))
    return float(rss)
```

3. minimize_GD Algorithm

Now, we will define function which minimizes least square error.
$arg_{min}||Y - XB|| = (Y - XB)^T(Y - XB)$ so our gradient descent method would be
$\beta_{i+1} = \beta_i + 2\mu X^T(Y - X\beta_i)$

In [42]:
```
def minimize_GD_algorithms(X,Y,beta_zero, mu, i_max):
    for i in range(0,i_max):
        beta_next=beta_zero+2*mu*np.matmul(np.transpose(X),Y-np.matmul(X,beta_zero))
        if least_square(X,Y,beta_next)>least_square(X,Y,beta_zero):
            return beta_next
        else:
            beta_zero = beta_next
    return beta_zero
```

4. Calculation of $|f(x_{i-1}) - f(x_i)|$ for Each Iteration

In this part, we would define a function that calculates $|f(x_{i-1}) - f(x_i)|$ and plot it against iteration number i.

```
In [43]: def iteration(X,Y,beta_zero, mu, i_max):
             iteration=[]
             for i in range(0,i_max):
                 beta_next=beta_zero+2*mu*np.matmul(np.transpose(X),Y-np.matmul(X,beta_zero))
                 iteration.append(abs(least_square(X,Y,beta_next)-least_square(X,Y,beta_zero)))
                 beta_zero = beta_next
             return iteration
```

4. RMSE

Now, we will define RMSE. $\sqrt{\frac{\sum_{q=1}^{T}(y_{test}^{q}-\tilde{y}^{q})^{2}}{T}}$

```
In [44]: #to be able use math.square() function
         import math
```

```
In [45]: def rmse(X,Y,beta):
             rmse_error_matrix = Y - X@beta
             total_error_rmse = 0
             for i in range(0,Y.shape[0]):
                 total_error_rmse = rmse_error_matrix[i]**2+total_error_rmse

             return math.sqrt(total_error_rmse/Y.shape[0])
```

5. Calculation of RMSE for Each Iteration

```
In [46]: def iteration_rmse(X_train,Y_train,beta_zero, mu, i_max, X_test, Y_test):
             iteration_rmse=[]
             for i in range(0,i_max):
                 beta_next=beta_zero+2*mu*np.matmul(np.transpose(X_train),Y_train-np.matmul(X_trai
                 iteration_rmse.append(rmse(X_test,Y_test,beta_next))
                 beta_zero = beta_next
             return iteration_rmse
```

B.2 AIRFARE AND DEMAND DATASET

B.2.1 Airfare and Demand: Creating X For Trainset

```
In [47]: #deleting the Price column from table and converting table to matrix
         x_airfare_train_notbias = np.array(airfare_demand_train.drop(['Price'],axis=1))

         #adding a 1 vector as column for bias
         x_airfare_train = np.hstack((x_airfare_train_notbias,np.ones((800,1))))

         #checking the dimensions of X matrix
         airfare_demand_train.shape
```

```
Out[47]: (800, 217)
```

B.2.2 Airfare and Demand: Creating Y For Trainset

In [48]:
```
#taking only price column and converting to vector matric
y_airfare_train =np.array(airfare_demand_train['Price']).reshape((800,1))

#checking the dimension of Y matrix
y_airfare_train.shape
```

Out[48]:
```
(800, 1)
```

B.2.3 Airfare and Demand: Creating X For Testset

In [49]:
```
#deleting the Price column from table and converting table to matrix
x_airfare_test_notbias = np.array(airfare_demand_test.drop(['Price'],axis=1))

#adding a 1 vector as column for bias
x_airfare_test = np.hstack((x_airfare_test_notbias,np.ones((200,1)
                                                          )))

#checking the dimensions of X matrix
airfare_demand_test.shape
```

Out[49]:
```
(200, 217)
```

B.2.4 Airfare and Demand: Creating Y For Testset

In [50]:
```
#taking only price column and converting to vector matric
y_airfare_test =np.array(airfare_demand_test['Price']).reshape((200,1))

#checking the dimension of Y matrix
y_airfare_test.shape
```

Out[50]:
```
(200, 1)
```

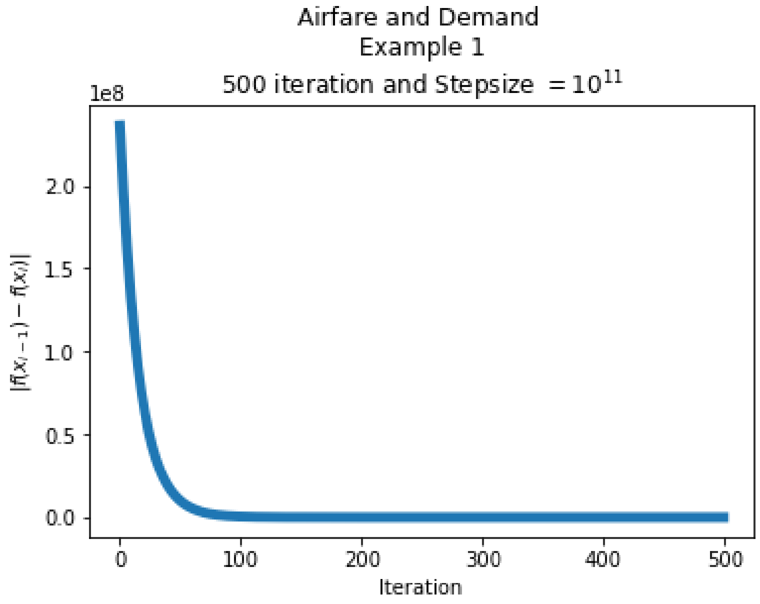
B.2.5 Airfare and Demand: Minimize GD Algorithm

Example 1: stepsize=10^{11} and 500 iteration

In [51]:
```
i_errors_1=iteration(x_airfare_train,y_airfare_train,np.ones((217,1)),0.00000000001,500)
i_errors_1
```

Out[51]:
```
[236185601.50268936,
 221635020.62460184,
 207981397.07751656,
 195169429.47863913,
 183147226.1803522,
 171866095.0322957,
 161280346.1063714,
 151347106.5853982,
 142026147.06542158,
 133279718.5679574,
 125072399.60177064,
 117370952.65458274,
 110144189.53326106,
 103362845.00690627,
 96999458.24089384,
 91028261.541502,
 85425075.96034431,
 80167213.33567238,
 75233384.37363124,
```

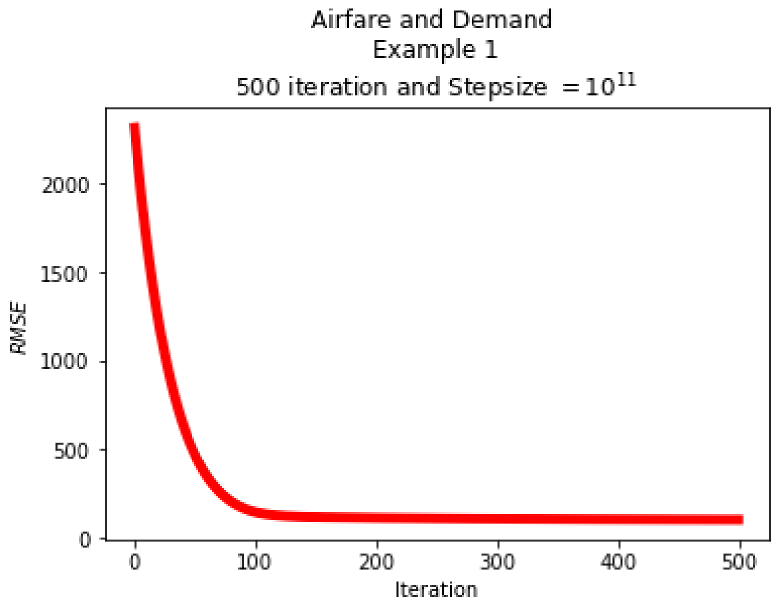
In [52]:
```
plt.title('Airfare and Demand \nExample 1\n$500$ iteration and Stepsize $=10^{11}$')
plt.plot(list(range(0,len(i_errors_1))),i_errors_1,linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('$|f(x_{i-1})-f(x_i)|$')
plt.show()
```



In [53]:
```
#we have to apply betas on test sets
i_errors_1_rmse=iteration_rmse(x_airfare_train,y_airfare_train,np.ones((217,1)),0.000000(
i_errors_1_rmse
```

Out[53]:
```
[2314.861205349582,
 2241.639736039368,
 2170.7248141192226,
 2102.044431042809,
 2035.5288412379234,
 1971.1104914533698,
 1908.7239523245898,
 1848.3058520885777,
 1789.7948123807519,
 1733.1313860485518,
 1678.2579969185322,
 1625.1188814556945,
 1573.6600322556599,
 1523.8291433121244,
 1475.5755570037938,
 1428.8502127466854,
 1383.605597259346,
 1339.795696390091,
 1297.3759484569123,
```

In [54]:
```
plt.title('Airfare and Demand \nExample 1\n$500$ iteration and Stepsize $=10^{11}$')
plt.plot(list(range(0,len(i_errors_1_rmse))),i_errors_1_rmse,'r',linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('$RMSE$')
plt.show()
```

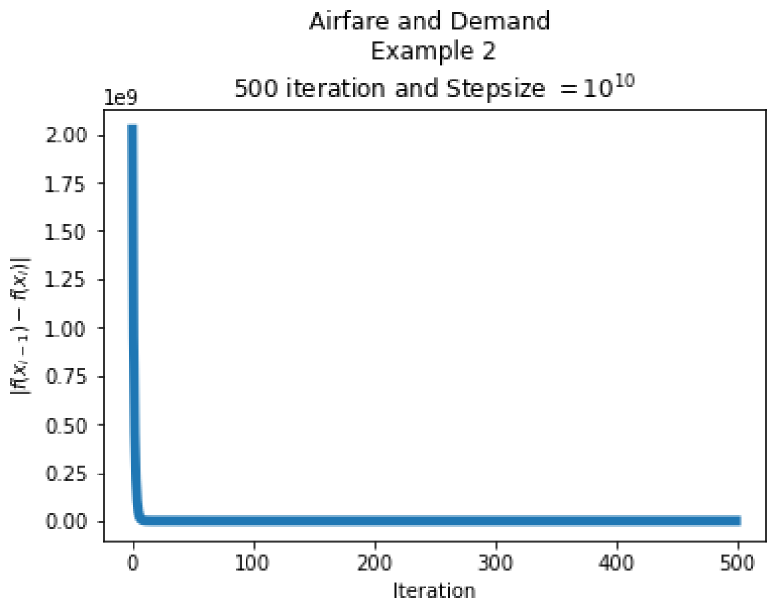


Example 2 : Stepsize = 10^{10} and 500 Iteration

In [55]:
```python
i_errors_2=iteration(x_airfare_train,y_airfare_train,np.ones((217,1)),0.0000000001,500)
i_errors_2
```

Out[55]:
```
[2023970739.5011024,
 955666627.2208867,
 451591719.987015,
 213689094.60754102,
 101361383.87401727,
 48285382.552323624,
 23173501.585682884,
 11264904.366633352,
 5594819.532621983,
 2876218.868001871,
 1557152.5387735479,
 904348.3397490103,
 570906.2401824873,
 392346.7132180631,
 290378.8366956152,
 227484.34509484097,
 185475.4041425176,
 155364.0173357036,
 132569.83179190196,
```

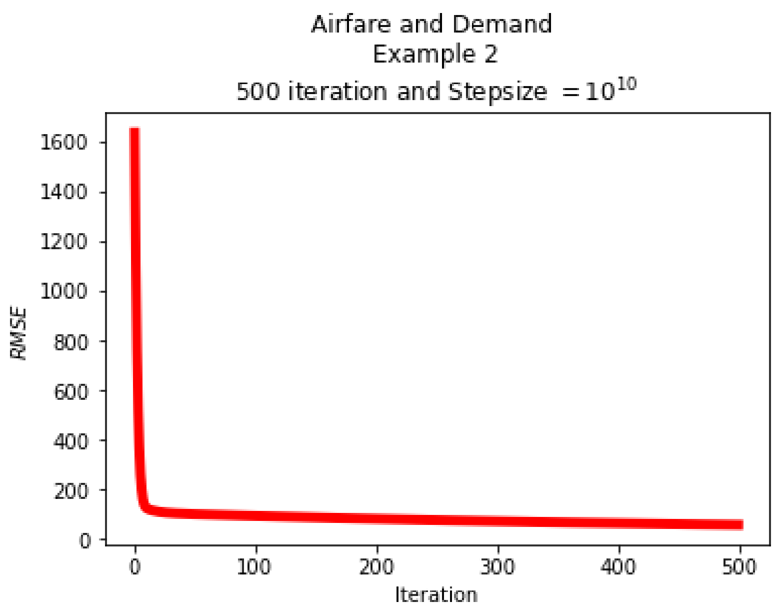
In [56]:
```python
plt.title('Airfare and Demand \nExample 2\n$500$ iteration and Stepsize $=10^{10}$')
plt.plot(list(range(0,len(i_errors_2))),i_errors_2,linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('$|f(x_{i-1})-f(x_i)|$')
plt.show()
```



In [57]:
```python
i_errors_2_rmse=iteration_rmse(x_airfare_train,y_airfare_train,np.ones((217,1)),0.0000000
i_errors_2_rmse
```

Out[57]:
```
[1635.2293188412818,
 1118.5414124668373,
 766.2794927861598,
 527.7683520168566,
 368.4603642246771,
 264.7807523952176,
 200.28855653901945,
 162.74912146782117,
 142.45287714569676,
 132.0221311438905,
 126.62784370210868,
 123.58635219466,
 121.58266673891364,
 120.02827500398102,
 118.67995835706141,
 117.44659182938067,
 116.29907384810612,
 115.23072163726354,
 114.24095775234024,
```

In [58]:
```python
plt.title('Airfare and Demand \nExample 2\n$500$ iteration and Stepsize $=10^{10}$')
plt.plot(list(range(0,len(i_errors_2_rmse))),i_errors_2_rmse,'r',linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('$RMSE$')
plt.show()
```

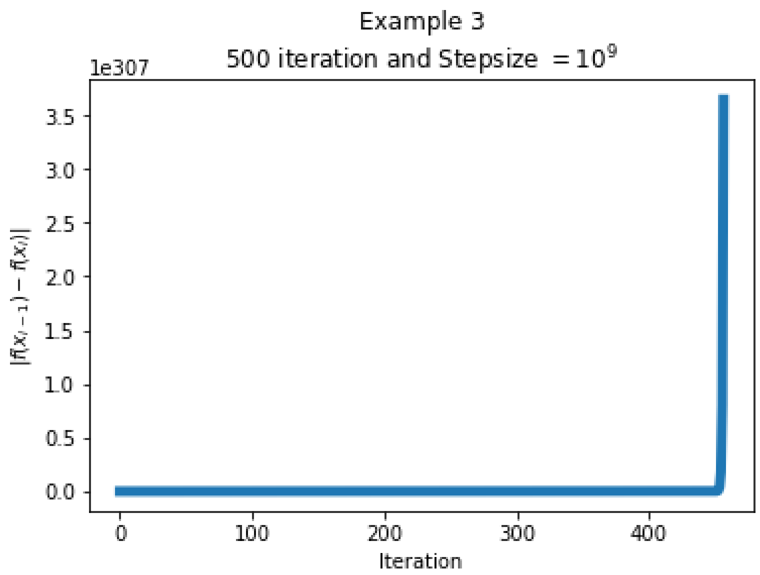


Example 3: Sepsize=0.1 and 100 Iteration

In [59]:
```
i_errors_3=iteration(x_airfare_train,y_airfare_train,np.zeros((217,1)),0.000000001,500)
i_errors_3
```

```
<ipython-input-41-91d310f79016>:2: RuntimeWarning: overflow encountered in matmul
  rss = ((Y-(X@B)).T)@(Y-(X@B))
```

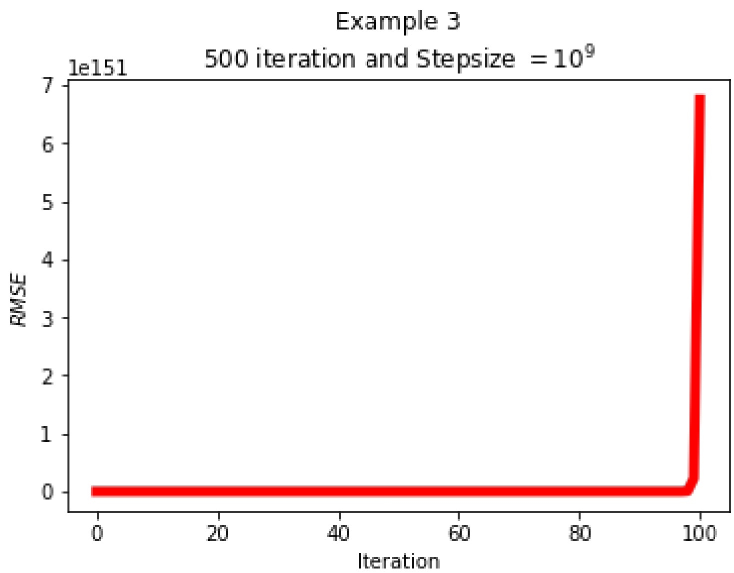
In [60]:
```
plt.title('Example 3\n$500$ iteration and Stepsize $=10^9$')
plt.plot(list(range(0,len(i_errors_3))),i_errors_3,linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('$|f(x_{i-1})-f(x_i)|$')
plt.show()
```



In [61]:
```
i_errors_3_rmse=iteration_rmse(x_airfare_train,y_airfare_train,np.zeros((217,1)),0.000000
i_errors_3_rmse
```

```
<ipython-input-45-effc3433c9bd>:5: RuntimeWarning: overflow encountered in add
  total_error_rmse = rmse_error_matrix[i]**2+total_error_rmse
<ipython-input-45-effc3433c9bd>:5: RuntimeWarning: overflow encountered in square
  total_error_rmse = rmse_error_matrix[i]**2+total_error_rmse
<ipython-input-46-e6ba670cf7a1>:4: RuntimeWarning: overflow encountered in matmul
  beta_next=beta_zero+2*mu*np.matmul(np.transpose(X_train),Y_train-np.matmul(X_train,
beta_zero))
<ipython-input-46-e6ba670cf7a1>:4: RuntimeWarning: invalid value encountered in matmu
l
  beta_next=beta_zero+2*mu*np.matmul(np.transpose(X_train),Y_train-np.matmul(X_train,
beta_zero))
<ipython-input-46-e6ba670cf7a1>:4: RuntimeWarning: invalid value encountered in add
  beta_next=beta_zero+2*mu*np.matmul(np.transpose(X_train),Y_train-np.matmul(X_train,
beta_zero))
```

In [62]:
```
plt.title('Example 3\n$500$ iteration and Stepsize $=10^9$')
plt.plot(list(range(0,len(i_errors_3_rmse))),i_errors_3_rmse,'r',linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('$RMSE$')
plt.show()
```



B.3. WINE QUALITY DATASET

B.3.1 Wine Quality: Creating X For Trainset

In [63]:
```
x_wine_without_bias=np.array(wine_quality_train.drop(['quality'],axis=1))

#adding a 1 vector as column for bias
x_wine_train = np.hstack((x_wine_without_bias,np.ones((1280,1))))

#checking the dimension
print(x_wine_train.shape)
```

```
(1280, 12)
```

B.3.2 Wine Quality: Creating Y For Trainset

In [64]:
```
y_wine_train = np.array(wine_quality_train['quality']).reshape((1280,1))

#checking the dimension
y_wine_train.shape
```

Out[64]:
```
(1280, 1)
```

B.3.3 Wine Quality: Creating X For Testset

In [65]:
```
x_wine_without_bias_test=np.array(wine_quality_test.drop(['quality'],axis=1))

#adding a 1 vector as column for bias
x_wine_test = np.hstack((x_wine_without_bias_test,np.ones((319,1))))

#checking the dimension
print(x_wine_test.shape)
```

```
(319, 12)
```

B.3.4 Wine Quality: Creating Y For Testset

In [66]:
```
y_wine_test = np.array(wine_quality_test['quality']).reshape((319,1))

#checking the dimension
y_wine_test.shape
```

Out[66]:
```
(319, 1)
```

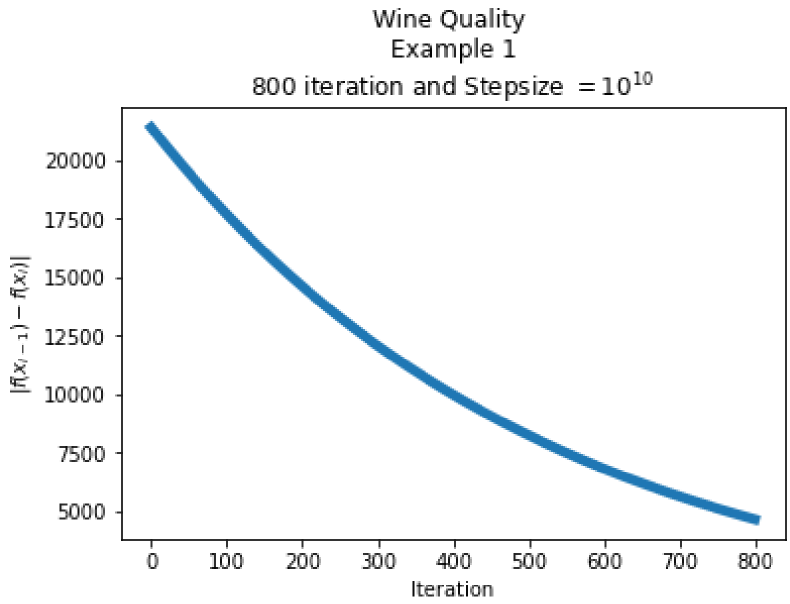
B.3.5 Wine Quality: Minimize GD Algorithm

Example 1: Stepsize = 10^{10} **amd 800 iterations**

In [67]:
```
j_errors_1=iteration(x_wine_train,y_wine_train,np.ones((12,1)),0.0000000001,800)
j_errors_1
```

Out[67]:
```
[21383.891377203166,
 21343.050162516534,
 21302.28698765859,
 21261.601703491062,
 21220.994161177427,
 21180.464212171733,
 21140.011708213016,
 21099.63650130853,
 21059.33844375424,
 21019.117388144135,
 20978.973187319934,
 20938.905694447458,
 20898.914762925357,
 20859.000246483833,
 20819.161999091506,
 20779.399875013158,
 20739.713728787377,
 20700.103415243328,
 20660.568789467216,
```

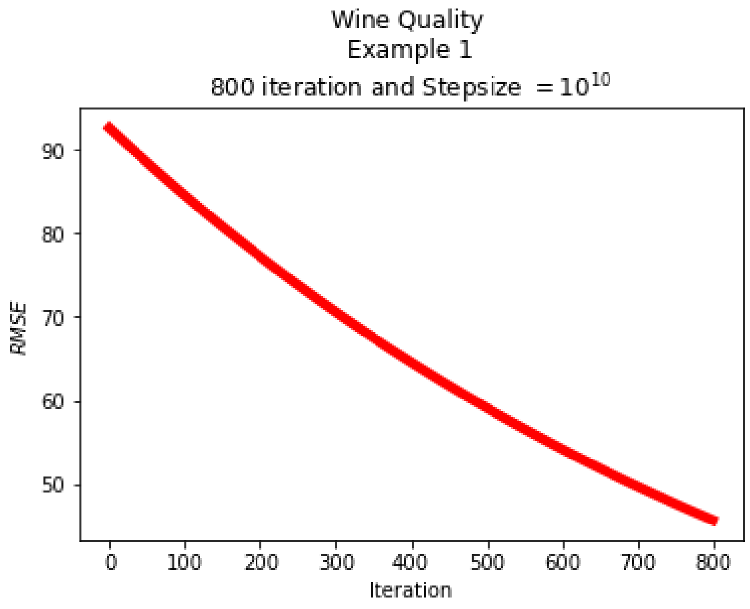
In [68]:
```python
plt.title('Wine Quality \nExample 1\n$800$ iteration and Stepsize $=10^{10}$')
plt.plot(list(range(0,len(j_errors_1))),j_errors_1,linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('$|f(x_{i-1})-f(x_i)|$')
plt.show()
```



In [69]:
```python
# we have to apply betas on test sets
j_errors_1_rmse=iteration_rmse(x_wine_train,y_wine_train,np.ones((12,1)),0.0000000001,80
j_errors_1_rmse
```

Out[69]:
```
[92.56928320557803,
 92.48464315264029,
 92.40008529236307,
 92.31560954723894,
 92.23121583983524,
 92.14690409279407,
 92.06267422883168,
 91.97852617073923,
 91.8944598413821,
 91.81047516370009,
 91.7265720607072,
 91.6427504554917,
 91.55901027121595,
 91.4753514311165,
 91.39177385850356,
 91.30827747676165,
 91.22486220934888,
 91.14152797979735,
 91.05827471171261,
```

In [70]:
```
plt.title('Wine Quality \nExample 1\n$800$ iteration and Stepsize $=10^{10}$')
plt.plot(list(range(0,len(j_errors_1_rmse))),j_errors_1_rmse,'r',linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('$RMSE$')
plt.show()
```

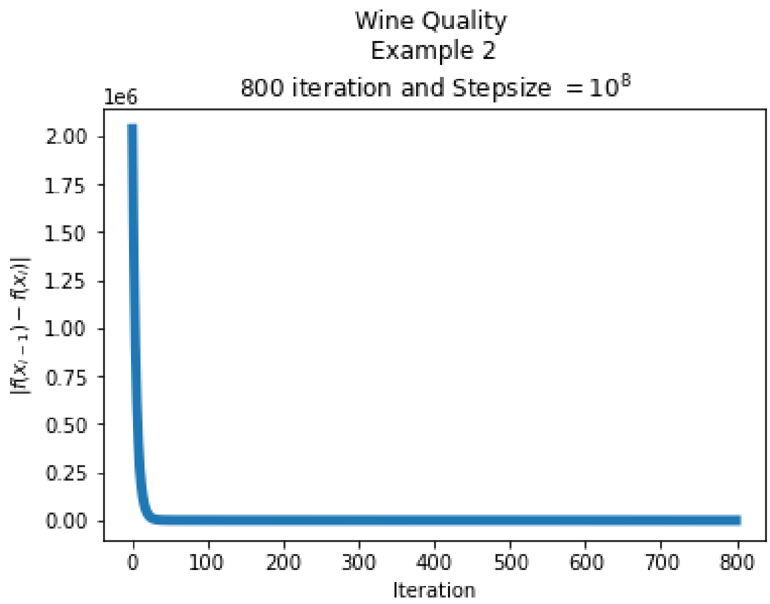


Example 2 Stepsize = 10^8 **and 800 iterations**

In [71]:
```
j_errors_2=iteration(x_wine_train,y_wine_train,np.ones((12,1)),0.00000001,800)
j_errors_2
```

Out[71]:
```
[2037210.4416177794,
 1666549.7481258418,
 1363362.4115338828,
 1115365.7419149112,
 912512.9617945272,
 746586.1858186652,
 610863.4934614212,
 499846.60704721464,
 409038.1426360635,
 334759.40964261466,
 274001.3777911649,
 224302.77370155836,
 183650.36849069363,
 150397.4167884572,
 123196.94292659068,
 100947.17155546218,
 82746.89194516605,
 67858.94766603195,
 55680.37252326199,
```

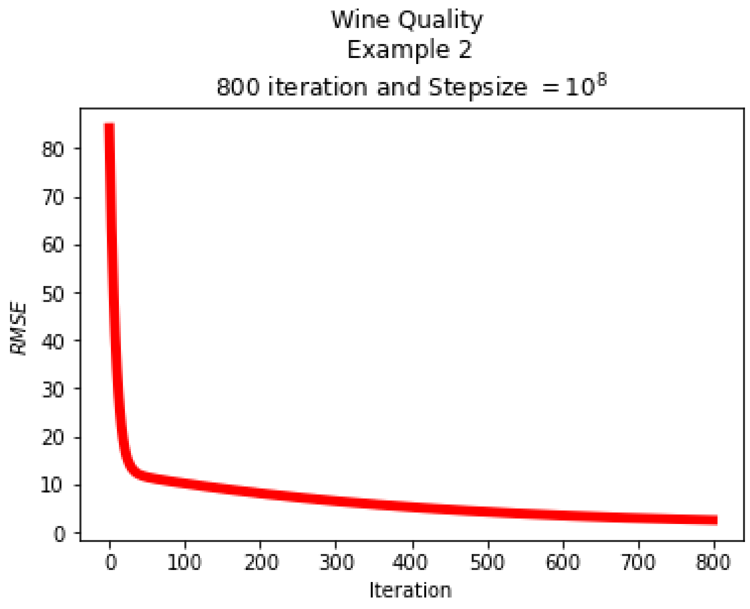
In [72]:
```
plt.title('Wine Quality \nExample 2\n$800$ iteration and Stepsize $=10^{8}$')
plt.plot(list(range(0,len(j_errors_2))),j_errors_2,linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('$|f(x_{i-1})-f(x_i)|$')
plt.show()
```



In [73]:
```
# we have to apply betas on test sets
j_errors_2_rmse=iteration_rmse(x_wine_train,y_wine_train,np.ones((12,1)),0.00000001,800,x
j_errors_2_rmse
```

Out[73]:
```
[84.18929255910629,
 76.54850644644527,
 69.65416892104872,
 63.43627162370663,
 57.83155393797647,
 52.782846234426906,
 48.238471635376456,
 44.15170011190655,
 40.48024944413772,
 37.185828337940514,
 34.233717817731154,
 31.59238791336663,
 29.233147612873434,
 27.129827010539834,
 25.258491449106128,
 23.597188104714945,
 22.125725744631428,
 20.825488170369564,
 19.67928108227814,
```

In [74]:
```
plt.title('Wine Quality \nExample 2\n$800$ iteration and Stepsize $=10^{8}$')
plt.plot(list(range(0,len(j_errors_2_rmse))),j_errors_2_rmse,'r',linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('$RMSE$')
plt.show()
```

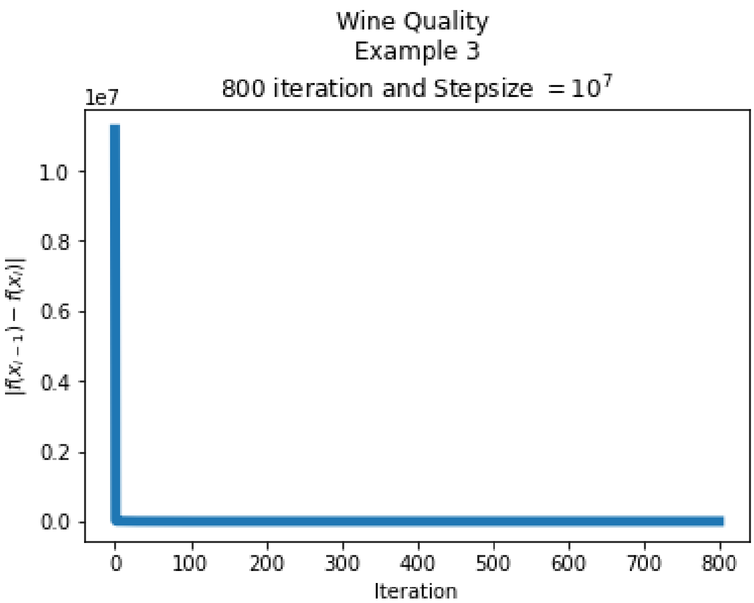


Example 3 Stepsize = 10^7 and 800 iterations

In [75]:
```python
j_errors_3=iteration(x_wine_train,y_wine_train,np.ones((12,1)),0.0000001,800)
j_errors_3
```

Out[75]:
```
[11174041.134120356,
 31768.870883507247,
 9621.342041634547,
 9147.183392298612,
 8734.7541205569,
 8341.085775971209,
 7965.248353591654,
 7606.432173089037,
 7263.864507131511,
 6936.807749236003,
 6624.5578163449245,
 6326.442624361982,
 6041.8206330302055,
 5770.079456999883,
 5510.634540072191,
 5262.92788974804,
 5026.4268693353515,
 4800.6230449987925,
 4585.0310852511175,
```

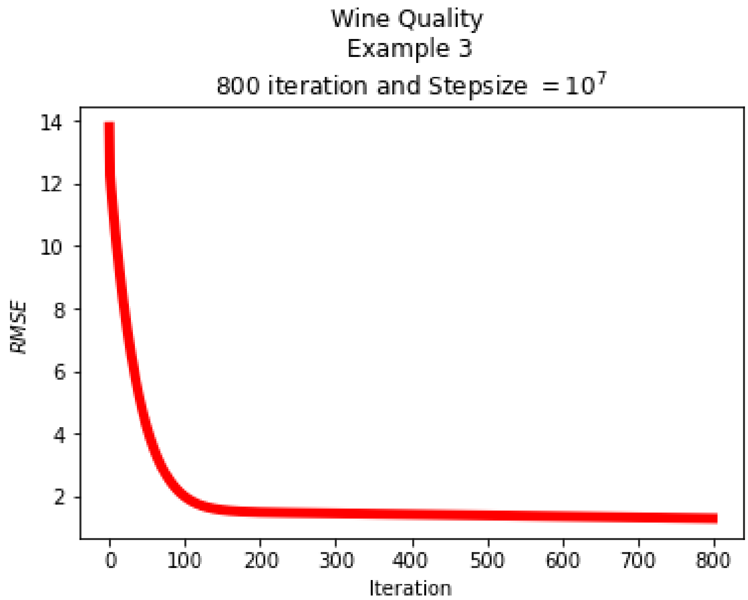
In [76]:
```python
plt.title('Wine Quality \nExample 3\n$800$ iteration and Stepsize $=10^{7}$')
plt.plot(list(range(0,len(j_errors_3))),j_errors_3,linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('$|f(x_{i-1})-f(x_i)|$')
plt.show()
```



In [77]:
```
# we have to apply betas on test sets
j_errors_3_rmse=iteration_rmse(x_wine_train,y_wine_train,np.ones((12,1)),0.0000001,800,x_
j_errors_3_rmse
```

Out[77]:
```
[13.808207744998537,
 12.253581025487737,
 11.94367892540877,
 11.671724480446514,
 11.407504567058377,
 11.149600324175852,
 10.897812959123751,
 10.652000422589587,
 10.412026272898236,
 10.177757355772686,
 9.94906363221713,
 9.725818102427116,
 9.50789673544986,
 9.295178400586476,
 9.087544800322048,
 8.88488040473685,
 8.687072387360315,
 8.494010562430228,
 8.305587323520399,
```

In [78]:
```
plt.title('Wine Quality \nExample 3\n$800$ iteration and Stepsize $=10^{7}$')
plt.plot(list(range(0,len(j_errors_2_rmse))),j_errors_3_rmse,'r',linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('$RMSE$')
plt.show()
```



B.4 PARKINSON

B.4.1 Parkinson: Creating X For Trainset

In [79]:
```
x_parkinson_without_bias=np.array(parkinson_train.drop(['Total_UPDRS'],axis=1))

#adding a 1 vector as column for bias
x_parkinson_train = np.hstack((x_parkinson_without_bias,np.ones((800,1))))

#checking the dimension
print(x_parkinson_train.shape)
```

```
(800, 217)
```

B.4.2 Parkinson: Creating Y For Trainset

In [80]:
```
y_parkinson_train = np.array(parkinson_train['Total_UPDRS']).reshape((800,1))

#checking the dimension
y_parkinson_train.shape
```

Out[80]:
```
(800, 1)
```

B.4.3 Parkinson: Creating X For Testset

In [81]:
```
x_parkinson_without_bias_test=np.array(parkinson_test.drop(['Total_UPDRS'],axis=1))

#adding a 1 vector as column for bias
x_parkinson_test = np.hstack((x_parkinson_without_bias_test,np.ones((200,1))))

#checking the dimension
print(x_parkinson_test.shape)
```

```
(200, 217)
```

In [82]:
```
x_parkinson_test.shape
```

Out[82]:
```
(200, 217)
```

B.4.4 Parkinson: Creating Y For Testset

In [83]:
```
y_parkinson_test = np.array(parkinson_test['Total_UPDRS']).reshape((200,1))

#checking the dimension
y_parkinson_test.shape
```

Out[83]:
```
(200, 1)
```

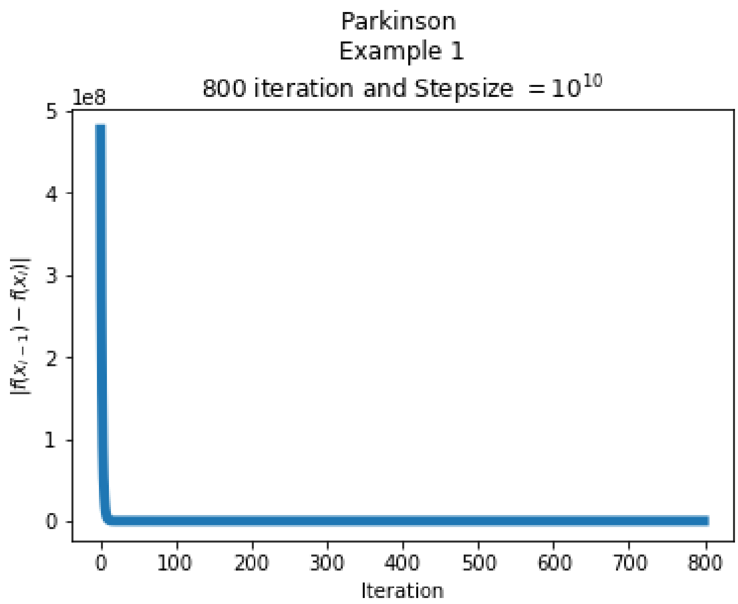
B.4.5 Parkinson: Minimize GD Algorithm

Example 1: Stepsize = 10^{10} amd 800 iterations

In [84]:
```
k_errors_1=iteration(x_parkinson_train,y_parkinson_train,np.ones((217,1)),0.0000000001,8
k_errors_1
```

Out[84]:
```
[477580376.4599302,
 274899979.7459488,
 158237313.3750422,
 91086374.87111378,
 52434342.21907973,
 30186254.067540884,
 17380261.59915811,
 10009130.929594934,
 5766301.858245552,
 3324120.419034004,
 1918390.1893042326,
 1109240.3988918662,
 643481.8299343586,
 375378.99727225304,
 221046.68712508678,
 132200.64599180222,
 81048.50091826916,
 51592.98686403036,
 34626.09224104881,
```

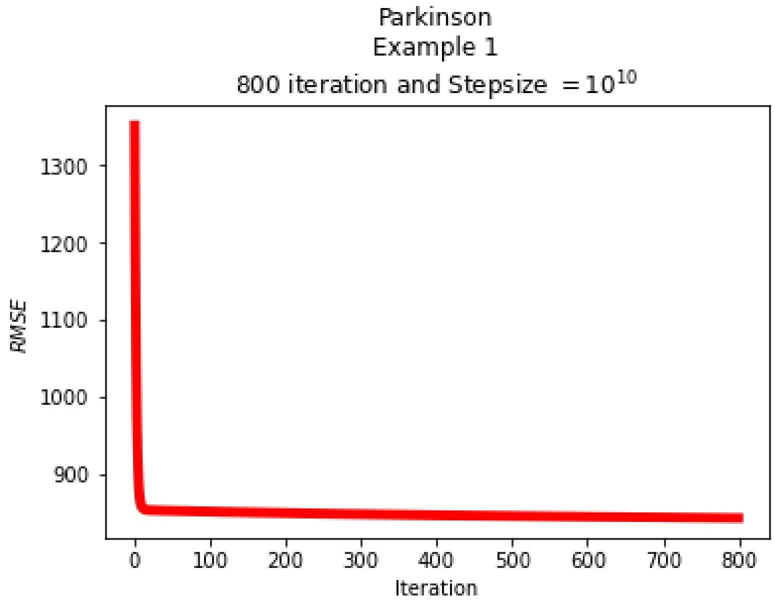
In [85]:
```
plt.title('Parkinson \nExample 1\n$800$ iteration and Stepsize $=10^{10}$')
plt.plot(list(range(0,len(k_errors_1))),k_errors_1,linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('$|f(x_{i-1})-f(x_i)|$')
plt.show()
```



In [86]:
```python
# we have to apply betas on test sets
k_errors_1_rmse=iteration_rmse(x_parkinson_train,y_parkinson_train,np.ones((217,1)),0.00
k_errors_1_rmse
```

Out[86]:
```
[1351.0743858011251,
 1168.756288739012,
 1048.8476337244695,
 972.5757252694862,
 925.3680921703916,
 896.7071302669666,
 879.5042449215197,
 869.2290720929811,
 863.0897897907334,
 859.4047580233109,
 857.1742100651478,
 855.807599094727,
 854.9567880401103,
 854.4162905926753,
 854.0643619634385,
 853.828439084158,
 853.6649205900042,
 853.5473382094776,
 853.4594246741215,
```

In [87]:
```python
plt.title('Parkinson\nExample 1\n$800$ iteration and Stepsize $=10^{10}$')
plt.plot(list(range(0,len(k_errors_1_rmse))),k_errors_1_rmse,'r',linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('$RMSE$')
plt.show()
```

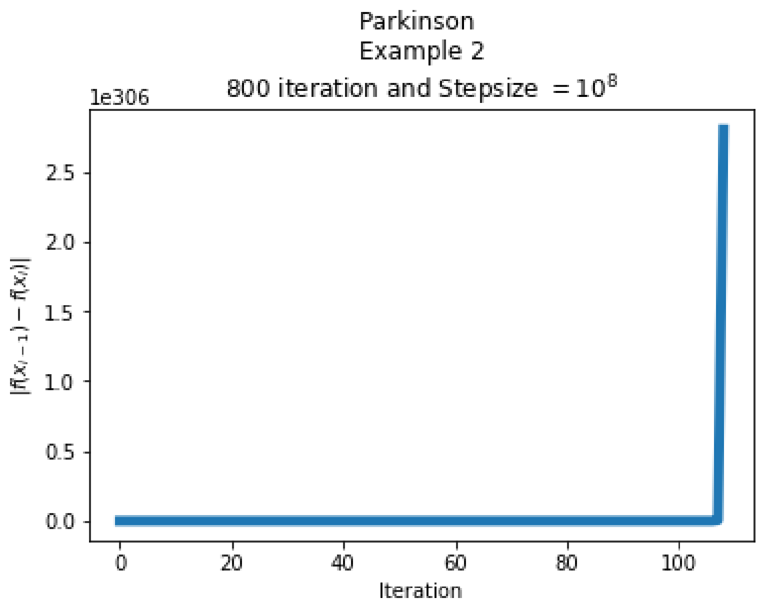


Example 2 Stepsize = 10^8 and 800 iterations

In [88]:
```
k_errors_2=iteration(x_parkinson_train,y_parkinson_train,np.ones((217,1)),0.00000001,800)
k_errors_2
```

```
<ipython-input-41-91d310f79016>:2: RuntimeWarning: overflow encountered in matmul
  rss = ((Y-(X@B)).T)@(Y-(X@B))
<ipython-input-43-aed54ee7d680>:4: RuntimeWarning: overflow encountered in matmul
  beta_next=beta_zero+2*mu*np.matmul(np.transpose(X),Y-np.matmul(X,beta_zero))
<ipython-input-43-aed54ee7d680>:4: RuntimeWarning: invalid value encountered in matmu
l
  beta_next=beta_zero+2*mu*np.matmul(np.transpose(X),Y-np.matmul(X,beta_zero))
<ipython-input-43-aed54ee7d680>:4: RuntimeWarning: invalid value encountered in add
  beta_next=beta_zero+2*mu*np.matmul(np.transpose(X),Y-np.matmul(X,beta_zero))
```

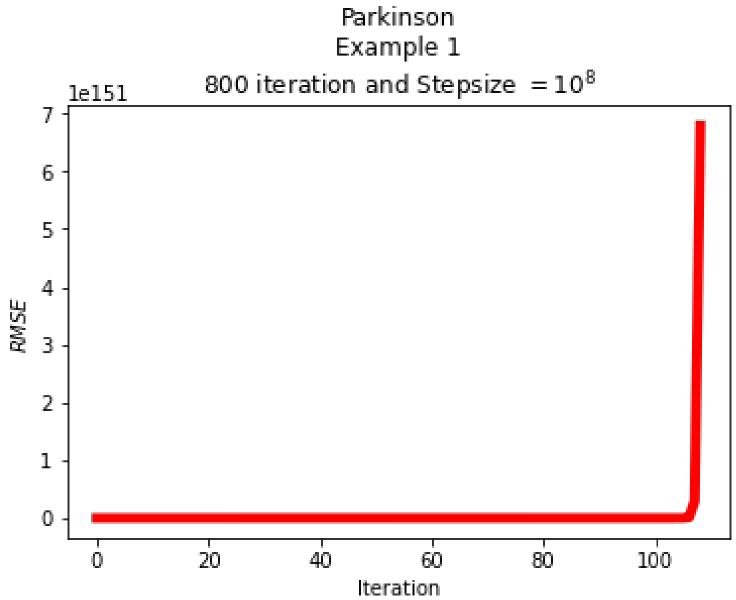
In [89]:
```
plt.title('Parkinson \nExample 2\n$800$ iteration and Stepsize $=10^{8}$')
plt.plot(list(range(0,len(k_errors_2))),k_errors_2,linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('$|f(x_{i-1})-f(x_i)|$')
plt.show()
```



In [90]:
```
# we have to apply betas on test sets
k_errors_2_rmse=iteration_rmse(x_parkinson_train,y_parkinson_train,np.ones((217,1)),0.00
k_errors_2_rmse
```

```
beta_zero))
<ipython-input-46-e6ba670cf7a1>:4: RuntimeWarning: invalid value encountered in add
  beta_next=beta_zero+2*mu*np.matmul(np.transpose(X_train),Y_train-np.matmul(X_train,
beta_zero))
```

In [91]:
```
plt.title('Parkinson\nExample 1\n$800$ iteration and Stepsize $=10^{8}$')
plt.plot(list(range(0,len(k_errors_2_rmse))),k_errors_2_rmse,'r',linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('$RMSE$')
plt.show()
```

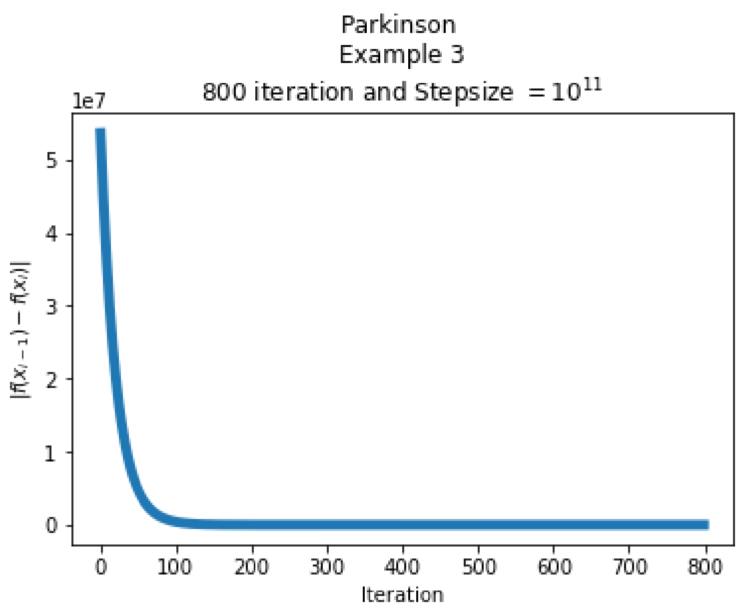


Example 3 Stepsize = 10^7 **and 800 iterations**

In [92]:
```
k_errors_3=iteration(x_parkinson_train,y_parkinson_train,np.ones((217,1)),0.00000000001,8
k_errors_3
```

```
 1103.1804255247116,
 1102.9127341508865,
 1102.6454337239265,
 1102.3785117864609,
 1102.1119558811188,
 1101.8457538485527,
 1101.5798952579498,
 1101.3143690228462,
 1101.0491651296616,
 1100.7842742204666,
 1100.5196867585182,
 1100.2553942203522,
 1099.9913884997368,
 1099.7276610732079,
 1099.4642048478127,
 1099.2010125517845,
 1098.9380772709846,
 1098.6753923892975,
 1098.4129519462585,
 1098.1507499217987,
```

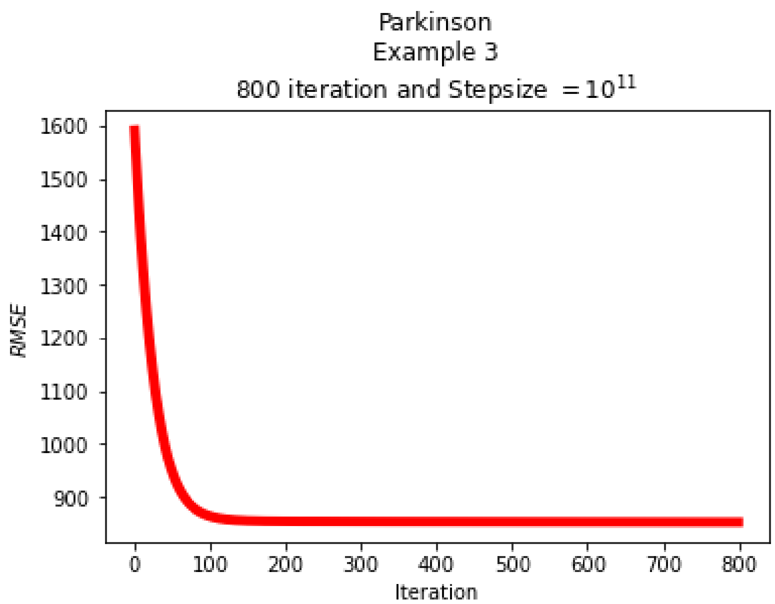
In [93]:
```
k_errors_3=iteration(x_parkinson_train,y_parkinson_train,np.ones((217,1)),0.00000000001,8
k_errors_3

plt.title('Parkinson \nExample 3\n$800$ iteration and Stepsize $=10^{11}$')
plt.plot(list(range(0,len(k_errors_3))),k_errors_3,linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('$|f(x_{i-1})-f(x_i)|$')
plt.show()
```



In [94]:
```
ave to apply betas on test sets
rs_3_rmse=iteration_rmse(x_parkinson_train,y_parkinson_train,np.ones((217,1)),0.0000000000
rs_3_rmse
```

Out[94]:
```
[1590.915664892526,
 1563.9936836649933,
 1537.9094057425746,
 1512.6429162733084,
 1488.1746382936972,
 1464.4853217308948,
 1441.5560331102222,
 1419.3681459787294,
 1397.9033320525893,
 1377.1435530927174,
 1357.0710535094681,
 1337.668353693332,
 1318.9182440645222,
 1300.8037798300925,
 1283.308276432953,
 1266.4153056727698,
 1250.108692474505,
 1234.3725122761198,
 1219.1910890030379,
```

In [95]:
```
plt.title('Parkinson\nExample 3\n$800$ iteration and Stepsize $=10^{11}$')
plt.plot(list(range(0,len(k_errors_3_rmse))),k_errors_3_rmse,'r',linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('$RMSE$')
plt.show()
```



EXERCISE 3: STEPLENGTH CONRTOL

3.1. Defining the Functions

3.1.1. Least Square

In [96]:
```
def least_square(X,Y,B):
    rss = ((Y-(X@B)).T)@(Y-(X@B))
    return float(rss)
```

3.1.2 Gradient Of Least Square

In [97]:
```
def grad_least_square(X,Y,B):
    return (-2*X.T)@(Y-X@B)
```

3.1.3. Backtracking Line Search

In [98]:
```
def stepsize_backtracking(X,Y,B,a,b):
    mu =1
    k = B- mu*grad_least_square(X,Y,B)
    if least_square(X,Y,k)>least_square(X,Y,B)-a*mu*grad_least_square(X,Y,B).T@grad_least
        return b*mu
    else:
        return 1
```

3.1.4 GD with Backtracking Line

In [99]:
```
def gd_stepsize(X,Y,B,i_max,a,b):
    error_list=[]
    mu=1
    for i in range(0,i_max):
        B_next= B - (a*mu)*grad_least_square(X,Y,B)
        error_list.append(abs(least_square(X,Y,B_next)-least_square(X,Y,B)))
        mu = stepsize_backtracking(X,Y,B_next,a,b)
        B = B_next
    return error_list
```

3.1.4 GD with Backtracking Line RMSE

In [100]:
```
def gd_stepsize_rmse(X,Y,B,i_max,a,b):
    error_list=[]
    mu=1
    for i in range(0,i_max):
        B_next= B - (a*mu)*grad_least_square(X,Y,B)
        error_list.append(rmse(X,Y,B_next))
        mu = stepsize_backtracking(X,Y,B_next,a,b)
        B = B_next
    return error_list
```

3.1.5 Bold Driver Step Size

In [101]:
```
def bold_driver(X,Y,B,mu,mu_plus,mu_negative):
    if least_square(X,Y,B)<=least_square(X,Y,B):
        mu = mu*mu_negative
    else:
        mu = mu*mu_plus
    return mu
```

3.1.6 GD with BOld Dirver Stepsize

```
In [102]: def gd_bold_driver(X,Y,B,i_max,mu_old,mu_plus,mu_minus):
    l = []
    mu = mu_old

    for i in range(0,i_max):
        B_next = B - mu* grad_least_square(X,Y,B)
        l.append(least_square(X,Y,B_next))
        B=B_next
        if least_square(X,Y,B)<=least_square(X,Y,B_next):
            mu = mu*mu_minus
        else:
            mu = mu*mu_plus
    return l
```

3.16 GD with Bold Driver Stepsize (RMSE)

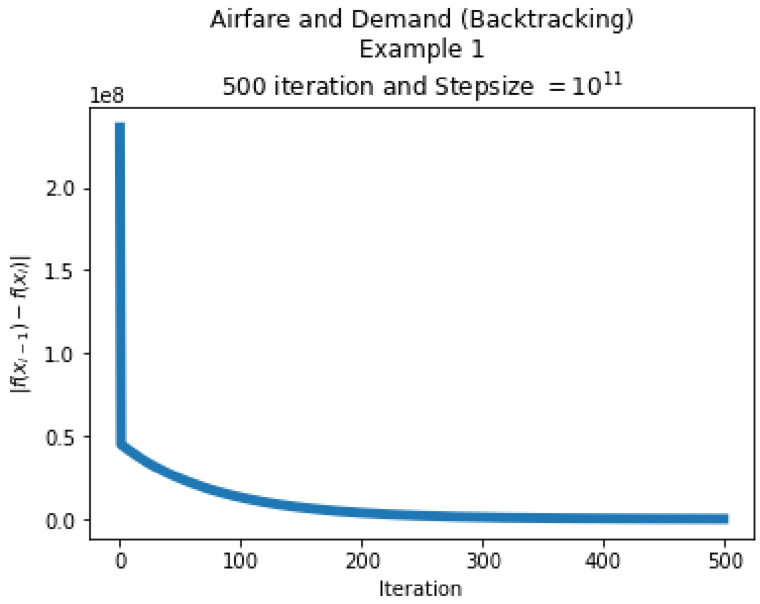
```
In [103]: def gd_bold_driver_rmse(X,Y,B,i_max,mu_old,mu_plus,mu_minus):
    l = []
    mu = mu_old

    for i in range(0,i_max):
        B_next = B - mu* grad_least_square(X,Y,B)
        l.append(rmse(X,Y,B_next))
        B=B_next
        if least_square(X,Y,B)<=least_square(X,Y,B_next):
            mu = mu*mu_minus
        else:
            mu = mu*mu_plus
    return l
```

4. AIRFARE AND DEMAND

Example 1: stepsize=10^{11} and 500 iteration Backtracking Line

```
In [104]: ss_errors_1=gd_stepsize(x_airfare_train,y_airfare_train,np.ones((217,1)),500,0.0000000000
ss_errors_1

Out[104]: [236185601.50268936,
 44890660.18761253,
 44330528.458440304,
 43777390.33042765,
 43231158.46886635,
 42691746.62971449,
 42159069.64598179,
 41633043.414280415,
 41113584.88153267,
 40600612.03186607,
 40094043.87365723,
 39593800.426730156,
 39099802.709739685,
 38611972.727684975,
 38130233.459599495,
 37654508.84638643,
 37184723.77879906,
 36720804.0855875,
 36262676.52177858,
```

In [105]:
```
plt.title('Airfare and Demand (Backtracking)\nExample 1\n$500$ iteration and Stepsize $=1
plt.plot(list(range(0,len(ss_errors_1))),ss_errors_1,linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('$|f(x_{i-1})-f(x_i)|$')
plt.show()
```

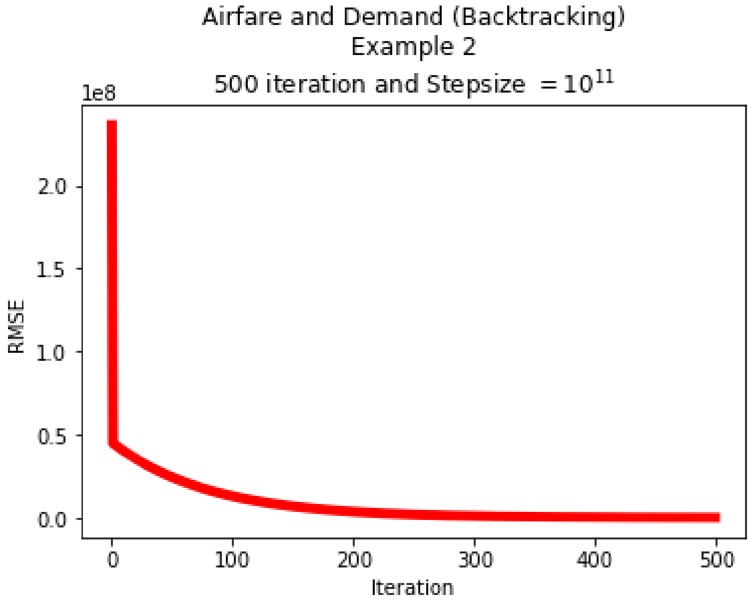


Example 2 stepsize=10^{11} and 500 iteration Backtracking Line With RMSE

In [106]:
```
ss_errors_2=gd_stepsize_rmse(x_airfare_train,y_airfare_train,np.ones((217,1)),500,0.00000
ss_errors_2
```

Out[106]:
```
[2126.0309019143538,
 2112.7929549911523,
 2099.6382807729524,
 2086.5663593986574,
 2073.5766742727606,
 2060.6687120449365,
 2047.841962589754,
 2035.0959189865255,
 2022.430077499268,
 2009.8439375568105,
 1997.3370017330055,
 1984.9087757270793,
 1972.5587683441008,
 1960.2864914755687,
 1948.0914600801204,
 1935.9731921643745,
 1923.9312087638712,
 1911.9650339241628,
 1900.0741946819883,
```

```
In [107]: plt.title('Airfare and Demand (Backtracking)\nExample 2\n$500$ iteration and Stepsize $=1
          plt.plot(list(range(0,len(ss_errors_1))),ss_errors_1,'r',linewidth=5)
          plt.xlabel('Iteration')
          plt.ylabel('RMSE')
          plt.show()
```

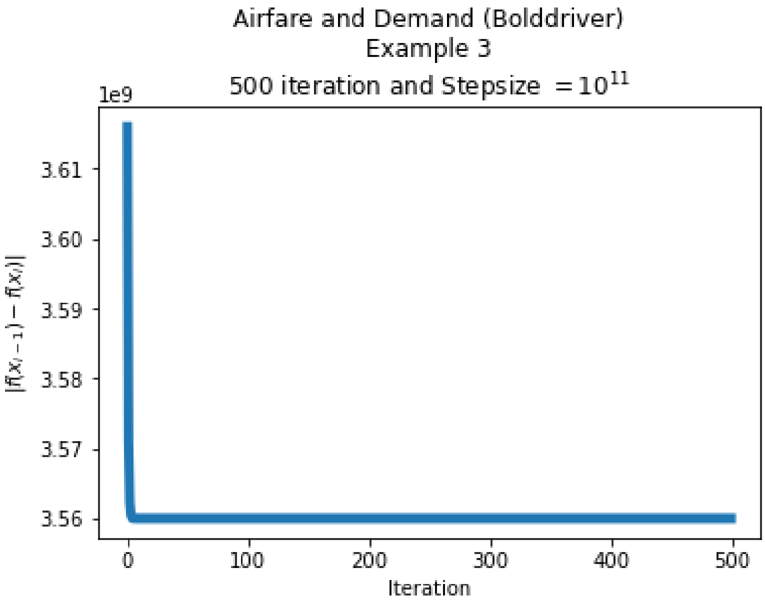


Example 3 stepsize=10^{11} and 500 iteration Bolddrive

```
In [108]: bd_errors_1=gd_bold_driver(x_airfare_train,y_airfare_train,np.ones((217,1)),500,0.000000
          bd_errors_1
```

```
Out[108]: [3616005916.715811,
           3571115256.5281982,
           3562226886.0984964,
           3560452771.024638,
           3560098090.1207666,
           3560027159.6224346,
           3560012973.7500505,
           3560010136.5846643,
           3560009569.1519513,
           3560009455.6654224,
           3560009432.968117,
           3560009428.428656,
           3560009427.5207634,
           3560009427.3391857,
           3560009427.3028703,
           3560009427.2956066,
           3560009427.294154,
           3560009427.2938643,
           3560009427.293806,
```

In [109]:
```python
plt.title('Airfare and Demand (Bolddriver)\nExample 3\n$500$ iteration and Stepsize $=10
plt.plot(list(range(0,len(bd_errors_1))),bd_errors_1,linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('$|f(x_{i-1})-f(x_i)|$')
plt.show()
```

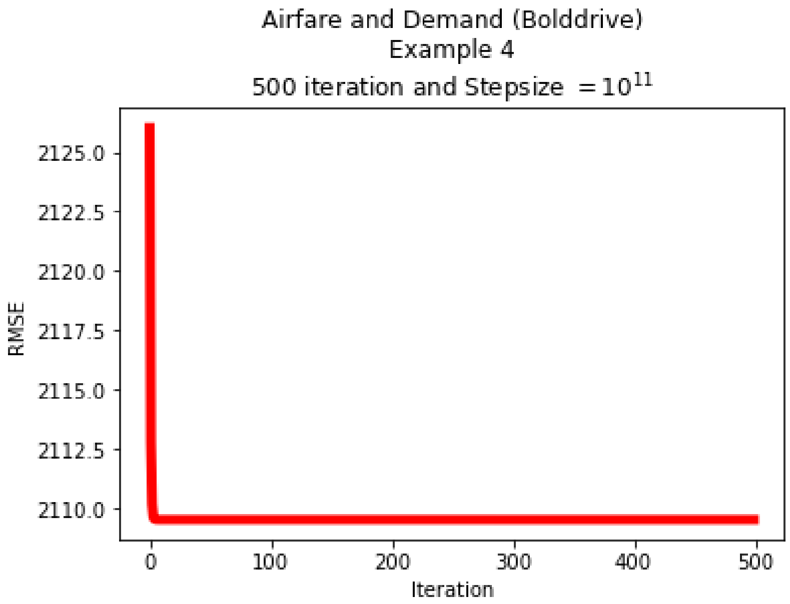


Example 4 stepsize=10^{11} and 500 iteration Bolddrive (RMSE)

In [110]:
```python
bd_errors_2=gd_bold_driver_rmse(x_airfare_train,y_airfare_train,np.ones((217,1)),500,0.0
bd_errors_2
```

Out[110]:
```
[2126.0309019143538,
 2112.7929549911523,
 2110.161986109863,
 2109.636452989187,
 2109.5313727581665,
 2109.5103577674226,
 2109.506154811491,
 2109.5053142219926,
 2109.505146104162,
 2109.5051124805973,
 2109.5051057558844,
 2109.505104410943,
 2109.5051041419533,
 2109.505104088157,
 2109.505104077396,
 2109.5051040752455,
 2109.5051040748144,
 2109.50510407473,
 2109.505104074711,
```

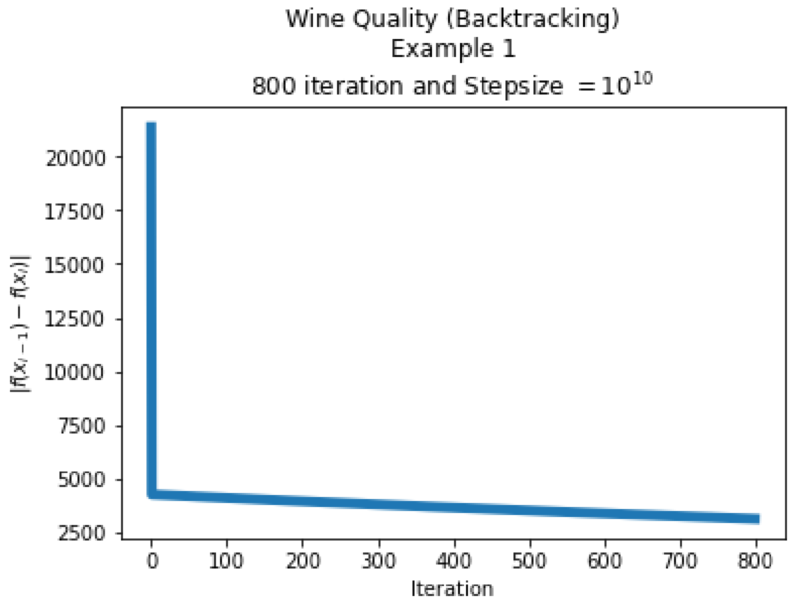
In [111]:
```
plt.title('Airfare and Demand (Bolddrive)\nExample 4\n$500$ iteration and Stepsize $=10^{
plt.plot(list(range(0,len(bd_errors_2))),bd_errors_2,'r',linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('RMSE')
plt.show()
```



5. WINE QUALITY

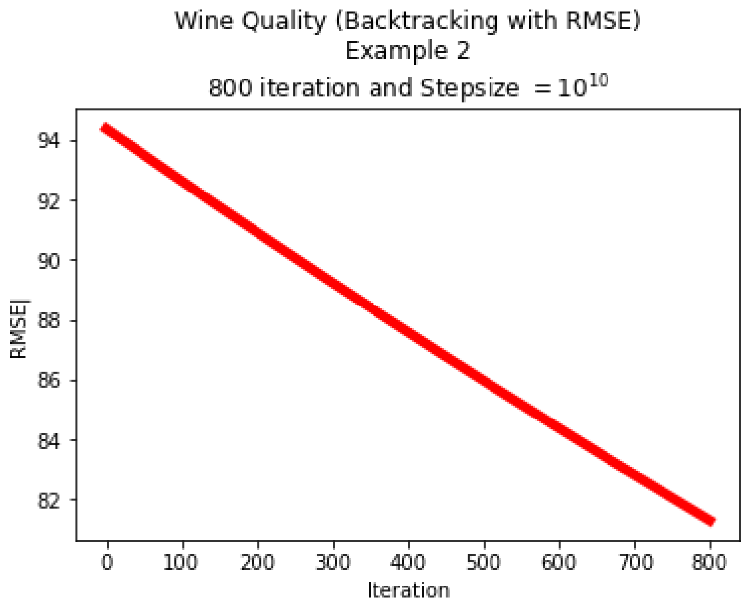
Example 1: 800 **iteration and Stepsize** $= 10^{10}$ **(Backtracking)**

In [112]:
```
jss_errors_1=gd_stepsize(x_wine_train,y_wine_train,np.ones((12,1)),800,0.0000000001,0.2)
jss_errors_1

plt.title('Wine Quality (Backtracking)\nExample 1\n$800$ iteration and Stepsize $=10^{10}
plt.plot(list(range(0,len(jss_errors_1))),jss_errors_1,linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('$|f(x_{i-1})-f(x_i)|$')
plt.show()
```



Example 2 800 **iteration and Stepsize** $= 10^{10}$ **(Backtracking with RMSE)**

In [120]:
```
jss_errors_2=gd_stepsize_rmse(x_wine_train,y_wine_train,np.ones((12,1)),800,0.0000000001
jss_errors_2

plt.title('Wine Quality (Backtracking with RMSE)\nExample 2\n$800$ iteration and Stepsize
plt.plot(list(range(0,len(jss_errors_2))),jss_errors_2,'r',linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('RMSE|')
plt.show()
```

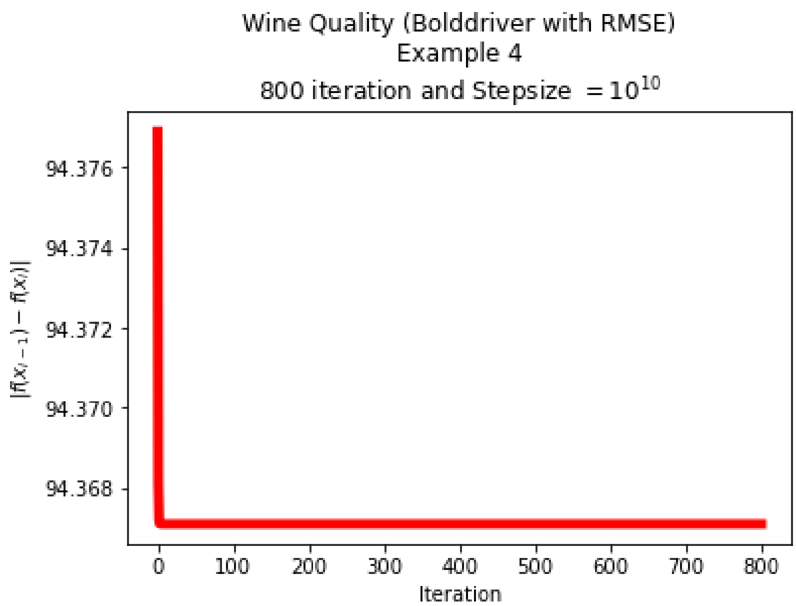


Example 3 800 **iteration and Stepsize** $= 10^{10}$ **Bolddriver**

In [114]:
```
jss_errors_3=gd_bold_driver(x_wine_train,y_wine_train,np.ones((12,1)),800,0.0000000001,1
jss_errors_3

plt.title('Wine Quality (Bolddriver)\nExample 3\n$800$ iteration and Stepsize $=10^{10}$
plt.plot(list(range(0,len(jss_errors_3))),jss_errors_3,linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('$|f(x_{i-1})-f(x_i)|$')
plt.show()
```

Example 3 800 **iteration and Stepsize** $= 10^{10}$ **Bolddriver with RMSE**

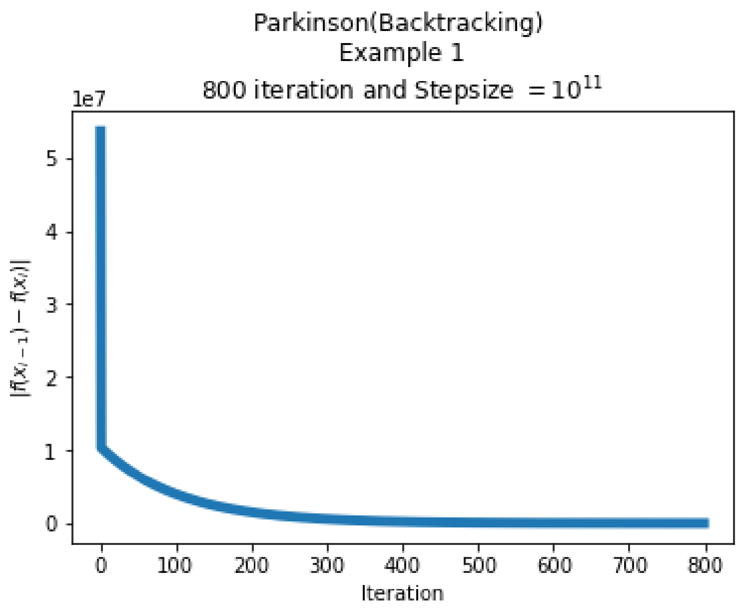
In [115]:
```
jss_errors_4=gd_bold_driver_rmse(x_wine_train,y_wine_train,np.ones((12,1)),800,0.00000000
jss_errors_4

plt.title('Wine Quality (Bolddriver with RMSE)\nExample 4\n$800$ iteration and Stepsize $
plt.plot(list(range(0,len(jss_errors_4))),jss_errors_4,'r',linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('$|f(x_{i-1})-f(x_i)|$')
plt.show()
```



3. PARKINSON

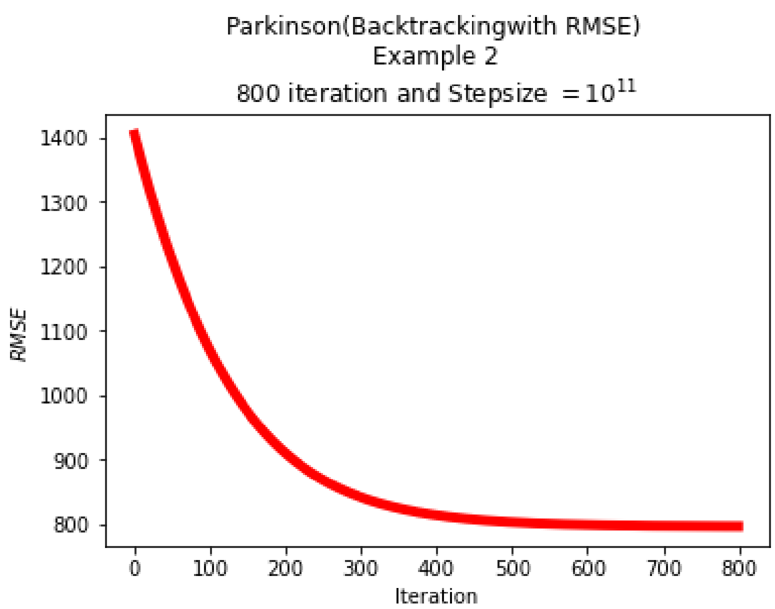
Example 1

In [116]:
```
kss_errors_1=gd_stepsize(x_parkinson_train,y_parkinson_train,np.ones((217,1)),800,0.00000
kss_errors_1

plt.title('Parkinson(Backtracking) \nExample 1\n$800$ iteration and Stepsize $=10^{11}$')
plt.plot(list(range(0,len(kss_errors_1))),kss_errors_1,linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('$|f(x_{i-1})-f(x_i)|$')
plt.show()
```



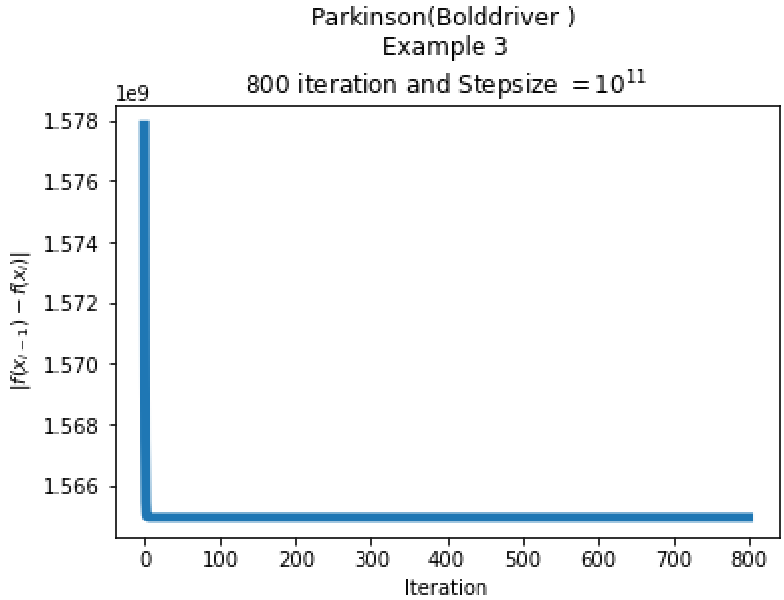
Example 2

In [117]:
```
kss_errors_2=gd_stepsize_rmse(x_parkinson_train,y_parkinson_train,np.ones((217,1)),800,0.
kss_errors_2

plt.title('Parkinson(Backtrackingwith RMSE) \nExample 2\n$800$ iteration and Stepsize $=1
plt.plot(list(range(0,len(kss_errors_2))),kss_errors_2,'r',linewidth=5)
plt.xlabel('Iteration')
plt.ylabel('$RMSE$')
plt.show()
```



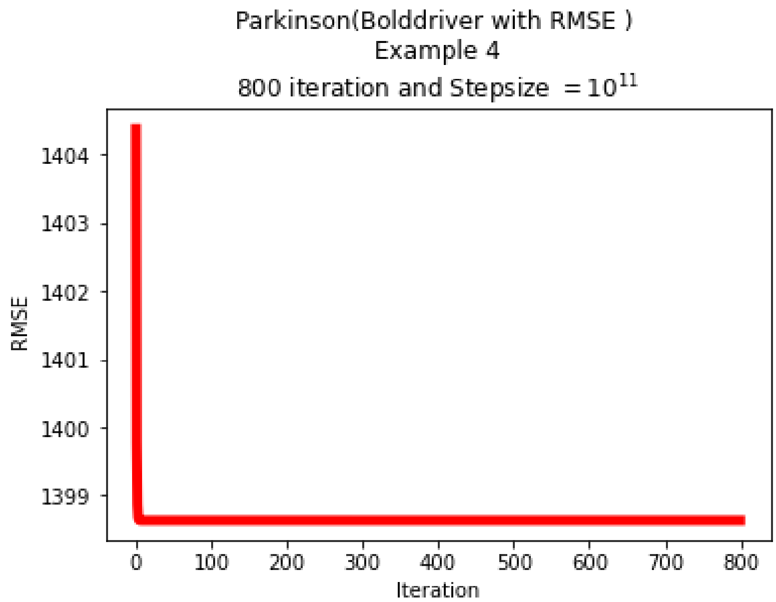
Example 3

In [118]:
```
kss_errors_3=gd_bold_driver(x_parkinson_train,y_parkinson_train,np.ones((217,1)),800,0.00
kss_errors_3

plt.title('Parkinson(Bolddriver ) \nExample 3\n$800$ iteration and Stepsize $=10^{11}$')
plt.plot(list(range(0,len(kss_errors_3))),kss_errors_3,linewidth=5)
plt.xlabel('Iteration')
plt.ylabel(' $|f(x_{i-1})-f(x_i)|$')
plt.show()
```



Example 4

In [119]:
```
kss_errors_4=gd_bold_driver_rmse(x_parkinson_train,y_parkinson_train,np.ones((217,1)),800
kss_errors_4

plt.title('Parkinson(Bolddriver with RMSE ) \nExample 4\n$800$ iteration and Stepsize $=1
plt.plot(list(range(0,len(kss_errors_4))),kss_errors_4,'r',linewidth=5)
plt.xlabel('Iteration')
plt.ylabel(' RMSE')
plt.show()
```



CONCLUSION

The learning and RMSE of Backtracking step length more drastic than Bold-driver step-length function (line is moreprendecular) RMSE for both of step-length functions are smoother than learning since it take into account square root of error

The logic for learning and RMSE of constant step-length is totally valid for backtrackingline and bold-driver step-lengths.

Gradient descent algorithm with backtrackingline and bold-driver methods learns quickly the dataa

In [ ]: