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

Nairobi Securities Exchange 20 Share Index (NSE20)

The Nairobi Securities Exchange 20 Share Index (NSE20) is a major stock market index. It tracks the performance of 20 best performing companies listed on the Nairobi Securities Exchange. These companies are selected based on a weighted market performance for a 12 month period. The performance is based on market capitalization, number of shares traded, number of deals and turnover.

COVID-19 in Kenya

The first case in Kenya was reported On 13 March. The Kenyan government identified and isolated a number of people who had come into contact with the first case. The daily positive cases in Kenya are on an upward trend with the highest daily count changing every now and then. COVID-19 has surely had an impact to the stock prices of the NSE20 index. I collected data since the first case was reported (13th March) to the date of preparing this report(21st July)

Load the downloaded NSE20 historical excel data (from https://www.investing.com/indices/kenya-nse-20-historical-data) into pandas. Use NSE20 = pd.read_excel(r"C:\Users.....pathname.....xlsx")

In [3]:
```
NSE20
```

Out[3]:

	Date	Price	Open	High	Low	Vol.	Change %
0	2020-03-12	1909.36	1909.36	1909.36	1909.36	-	0.0015
1	2020-03-13	1906.43	1906.43	1906.43	1906.43	-	-0.0037
2	2020-03-16	1913.57	1913.57	1913.57	1913.57	-	0.0105
3	2020-03-17	1893.65	1893.65	1893.65	1893.65	-	-0.0038
4	2020-03-18	1900.81	1900.81	1900.81	1900.81	-	-0.0002
...	...	...	...	...	...	...	...
82	2020-07-13	2048.87	2048.87	2048.87	2048.87	-	-0.0019
83	2020-07-14	2052.85	2052.85	2052.85	2052.85	-	-0.0025
84	2020-07-15	2057.96	2057.96	2057.96	2057.96	-	-0.0314
85	2020-07-16	2124.78	2124.78	2124.78	2124.78	-	-0.0501
86	2020-07-17	2236.81	2236.81	2236.81	2236.81	-	-0.0344

87 rows × 7 columns

In [4]:
```
NSE20.shape
```

Out[4]:
```
(87, 7)
```

We first create the start_date,end_date and removelist for the period of study. The remove list contains the Kenyan Public holidays that were in the study period. There were five holidays: Good Friday, Easter Monday,Labour day, Eid-al-Fitr and Madaraka day.

In [6]:
```
start_date = dt.date( 2020, 3, 12 )
end_date = dt.date( 2020, 7, 17 )
x=pd.bdate_range(start =start_date ,end = end_date)
removelist =['2020-04-10', '2020-04-13','2020-05-01','2020-05-25','2020-06-01']
x.shape
```

Out[6]:
```
(92,)
```

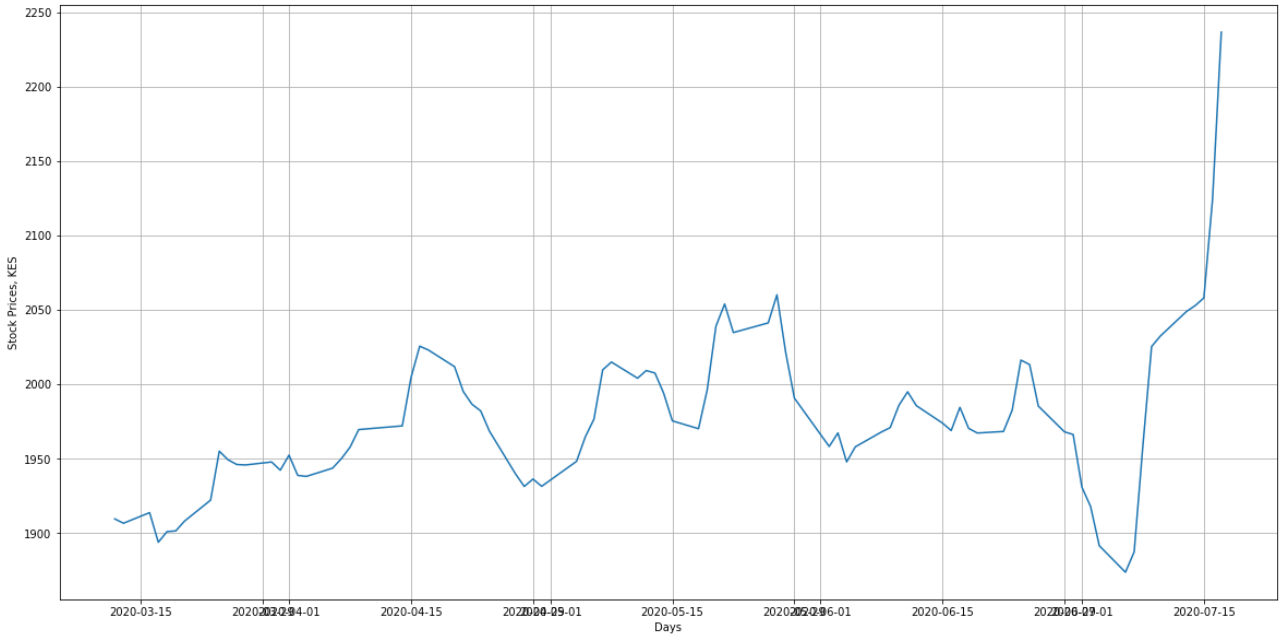
In [7]:
```
start_date = dt.date( 2020, 3, 12 )
end_date = dt.date( 2020, 7, 17 )
dates=pd.bdate_range(start =start_date ,end = end_date)
removeList = []
for each in dates:
    if each.month==4 and each.day ==10: #Good Friday
        removeList.append(each)
    if each.month==4 and each.day ==13:#Easter Monday
        removeList.append(each)
    if each.month==5 and each.day ==1: #Labour day
        removeList.append(each)
    if each.month==5 and each.day ==25: #Eid-al-Fitr
        removeList.append(each)
    if each.month==6 and each.day ==1:  #Madaraka day
        removeList.append(each)
dates = dates.drop(removeList)
len(dates)
```

Out[7]:
```
87
```

As seen above the dates dataframe originally contained 92 days but after removing the public holidays it contains 87days. This is equal to the number of days contained in the earlier NSE20 dataframe.

Next we plot the NSE20 open stock price to visualize the stock price movements during the period of the study.

In [44]:
```
plt.figure(figsize = (20, 10)) # create a figure object with (width, height) in
 inches
pos=np.arange(1,len(NSE20['Date'])+1)
x=dates
y=NSE20['Open']
plt.plot(x,y)
plt.grid()
plt.xlabel('Days')
plt.ylabel('Stock Prices, KES')
plt.show()
```



As seen above, the stock price movements are random, moving up and down from month to month.

We first use NSE20 June stock prices to make predictions for July

We want to test how efficient the Discrete-time Geometric Brownian Motion is in predicting the July NSE 20 stock price. We retrieve the June historical data and apply the GBM. Thereafter, we will make a plot to visualize the error between the predicted July NSE20 stock price and the actual July stock price. This will help us understand that the descrete-time GBM isn't a perfect stock price prediction tool. Though it is helful in giving us insights on the possible paths that the stock price can follow.

Geometric Brownian Motion (GBM) Parameter Definitions

The following are the notations for GBM that will be used throughout this study: So : initial stock price; d_t : Time increment (we will use a day in our case); T : Prediction time length(Prediction time points); N : Prediction time points (given by T/dt); t : Prediiction time points array for example [1, 2, 3, .. , N] ; mu : Mean daily returns; sigma : standard deviation daily returns; b : Brownian increments arrays; W : Brownian path array;

In [8]:
```
#Retrieve historical stock prices between our start_date and end_date.
start_date = pd.to_datetime('2020-06-01')
end_date = pd.to_datetime('2020-06-30')
pred_end_date = pd.to_datetime('2020-08-21')
```

```
NSE20['Date']= pd.to_datetime(NSE20['Date'],format='%y/%m/%d')
June=NSE20[NSE20['Date'].between(start_date,end_date)]
June['Date'] = June['Date'].astype('datetime64')
June_data=June.sort_values(by='Date', ascending = False)
June_data
```

```
C:\ProgramData\Anaconda\lib\site-packages\ipykernel_launcher.py:7: SettingWithCo
pyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stab
le/user_guide/indexing.html#returning-a-view-versus-a-copy
  import sys
```

Out[8]:

	Date	Price	Open	High	Low	Vol.	Change %
73	2020-06-30	1966.12	1966.12	1966.12	1966.12	-	0.0183
72	2020-06-29	1967.93	1967.93	1967.93	1967.93	-	0.0009
71	2020-06-26	1985.27	1985.27	1985.27	1985.27	-	0.0088
70	2020-06-25	2013.19	2013.19	2013.19	2013.19	-	0.0141
69	2020-06-24	2016.20	2016.20	2016.20	2016.20	-	0.0015
68	2020-06-23	1982.50	1982.50	1982.50	1982.50	-	-0.0167
67	2020-06-22	1968.19	1968.19	1968.19	1968.19	-	-0.0072
66	2020-06-19	1967.13	1967.13	1967.13	1967.13	-	-0.0005
65	2020-06-18	1970.26	1970.26	1970.26	1970.26	-	0.0016
64	2020-06-17	1984.40	1984.40	1984.40	1984.40	-	0.0072
63	2020-06-16	1968.78	1968.78	1968.78	1968.78	-	-0.0079
62	2020-06-15	1973.82	1973.82	1973.82	1973.82	-	0.0026
61	2020-06-12	1985.54	1985.54	1985.54	1985.54	-	0.0059
60	2020-06-11	1994.86	1994.86	1994.86	1994.86	-	0.0047
59	2020-06-10	1985.76	1985.76	1985.76	1985.76	-	-0.0046
58	2020-06-09	1970.79	1970.79	1970.79	1970.79	-	-0.0075
57	2020-06-08	1967.84	1967.84	1967.84	1967.84	-	-0.0015
56	2020-06-05	1957.86	1957.86	1957.86	1957.86	-	-0.0051
55	2020-06-04	1947.70	1947.70	1947.70	1947.70	-	-0.0052
54	2020-06-03	1967.19	1967.19	1967.19	1967.19	-	0.0100
53	2020-06-02	1958.07	1958.07	1958.07	1958.07	-	-0.0046

In [9]:
```
June_data.shape
```

Out[9]:
```
(21, 7)
```

We will use the initial stock price (So) as the closing stock price on June 30.

In [10]:
```
June_data.shape[0]+52 #locating the row with the June 30 closing stock price
```

Out[10]:
```
73
```

In [11]:
```
So = June_data.loc[June_data.shape[0]+52,"Price"] #[0] is used to index axis 0
 i.e rows
print(So)
```

```
1966.12
```

d_t is the model's time increment. It is represented in days. The daily closing prices of NSE20 stocksare used thus the model time increment is 1 day.

In [12]:
```
d_t = 1
print(d_t)
```

```
1
```

T denotes the prediction time horizon length. We use the June data to predict the July stock price. This report was made on July 21 thus the prediction end time is 21st July.

In [13]:
```
T=np.busday_count('2020-07-01', '2020-07-21')
T
```

Out[13]:
```
14
```

In [14]:
```
N = T/d_t
print(N)
```

```
14.0
```

t is an array that will start from 1 until 14. This is interpreted as: one day in historical data translates to one day in predictions.

In [15]:
```
t = np.arange(1, int(N) + 1)
print(t)
```

```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14]
```

mu is the mean daily returns based on the selected historical data. rk=(Sk - Sk_1)/(Sk_1) ;where k is time period , Sk is stock period at time k and Sk_1 is stock price after applying one day lag.

In [16]:
```
returns = (June_data['Price'].loc[72:] - June_data['Price'].shift(1).loc[72:]) /
June_data['Price'].shift(1).loc[72:]
print(returns)
```

```
72    0.000921
71    0.008811
70    0.014064
69    0.001495
68   -0.016715
67   -0.007218
66   -0.000539
65    0.001591
64    0.007177
63   -0.007871
62    0.002560
61    0.005938
60    0.004694
59   -0.004562
58   -0.007539
```

```
57   -0.001497
56   -0.005072
55   -0.005189
54    0.010007
53   -0.004636
Name: Price, dtype: float64
```

Just to visualize what happens when a one day lag is applied, look at the below data. You should realize that previously the rows began at row 73 but this time it has shifted down to 72.

In [68]:
```
June_data['Price'].shift(1).loc[72:]
```

Out[68]:
```
72    1966.12
71    1967.93
70    1985.27
69    2013.19
68    2016.20
67    1982.50
66    1968.19
65    1967.13
64    1970.26
63    1984.40
62    1968.78
61    1973.82
60    1985.54
59    1994.86
58    1985.76
57    1970.79
56    1967.84
55    1957.86
54    1947.70
53    1967.19
Name: Price, dtype: float64
```

Next, to obtain μ we need to calculate the arithmetic average of the returns. It will used thereafter in the models drift component. μ =(1/(|k|))*Σ(rk) From below, June has negative return on average which will be taken into account when calculating July predictions.

In [17]:
```
mu = np.mean(returns)
print(mu)
```

```
-0.0001790085400076574
```

Sigma represents the June returns standard deviation. It helps to incorporate random shocks into July predictions. Sigma determines the random shocks' magnitude. On its own Sigma doesn't add required randomness thus standard normal variable is used in picking random values. Thus the contribution of sigma is on scaling the magnitude of random shock to ensure that they occur in accordance with the historical stock prices volatility.

In [18]:
```
sigma = np.std(returns)
print(sigma)
```

```
0.0072236836688402166
```

The array b adds randomness to the model since it stores the random numbers generated from the standard normal distribution.
The numpy.random.normal() will be used to generate random values from the standard normal

distribution. In the model, two arrays with length 14 will be created to depict two different scenarios that the stock prices can take.

In [19]:
```
#generalizing the idea on developing random shocks
shocks=np.random.normal(0, 1, int(N))
shocks
```

Out[19]:
```
array([-1.40019014, -0.86898563,  0.02585634,  0.18993019, -0.07094578,
        1.53832246,  1.11825774,  0.00414566,  1.11521621,  0.51543267,
        1.93286481,  0.47020738,  0.36568351,  1.09776921])
```

In [20]:
```
#generate two sets of shocks and combine them into one code
times=2
b={str(num_times): np.random.normal(0, 1, int(N)) for num_times in range(1, time
s + 1)}
b
```

Out[20]:
```
{'1': array([ 1.05786261, -0.52961493, -1.71862859, -1.17537669,  0.70268112,
        -0.61520111,  0.18151524,  0.69821913, -1.20001982, -1.61388321,
        -2.0410876 , -0.06041706,  2.06832528, -1.25520666]),
 '2': array([-0.56257449, -1.10143339, -1.45637756, -0.5444432 ,  0.64499488,
         0.32458614, -1.41187082, -1.63290442, -0.45709989, -1.77287198,
        -0.23782408,  0.44398346, -0.17085925,  0.45197437])}
```

W stores the Brownian path. This path determines price fluctuations frpm initial stock price (So) to subsequent time points. To clearly interpret b, consider stock price at point 4 (predicted value S_4). To make next prediction, a random shock b(5) is applied to S_4. For W, it will be the entire path covered by all the random shocks from the start of prediction time horizon i.e. it is the cummulative effect. Therefore, it is the cumulative sum of array b elements.

In [21]:
```
W={str(num_times): b[str(num_times)].cumsum() for num_times in range(1, times +
1)}
W
```

Out[21]:
```
{'1': array([ 1.05786261,  0.52824767, -1.19038092, -2.36575761, -1.66307649,
        -2.27827761, -2.09676237, -1.39854324, -2.59856307, -4.21244628,
        -6.25353387, -6.31395094, -4.24562565, -5.50083232]),
 '2': array([-0.56257449, -1.66400788, -3.12038544, -3.66482863, -3.01983376,
        -2.69524761, -4.10711843, -5.74002285, -6.19712274, -7.96999472,
        -8.2078188 , -7.76383534, -7.93469459, -7.48272022])}
```

NOTE THAT: When predicting the price at time point k given that we are at time point k_1, the stock price must obey the long term trend as it gets exposed to a random shock.

Drift and Diffusion GBM

Drift incorporates the stock price longer-term trend. Diffusion reflects shorter-term random fluctuations. The array b stores the retrieved random shock information. This is retrieved from standard normal random variable z. We multiply the random value z(k) with sigma to get the diffusion component. YOU'VE NOW NOTICED how the randomness is incorporated into the GBM model. Diffusion component helps create as many scenarios as we want since it involves Wiener process(from definition it's created from independent, stationary and normally distributed random shocks). drift_k=μ-0.5σ**2 *diff_k= σbk = σZk Sk=Sk_1*exp(drift_k + diff_k) = Sk_1*exp(μ-0.5*σ**2+ σZk)* **Sk=Sk_2*exp(μ-0.5σ*2+ σZk_1)*exp(μ-0.5σ**2+ σZk)** **Sk=Sk_3*exp(μ-0.5σ*2+ σZk_2)*exp(μ-0.5σ**2+ σZk_1)*exp(μ-0.5σ*2+ σZk) .....and so on. Generally;

Sk=So*exp((μ-0.5σ*2)*k + σΣZi)* FROM ABOVE we can predict S(k) from So. This involves adding the k many drifts effect and the cumulative diffusion up to k.

The above translates into a single step predictions expression:
Sk=So*exp((mu-0.5σ*2)*tk + σWk)

Making predictions

First, we calculate drift for all the prediction time points. Do this using array t by just multiplying it with drift. This generates an array of drifts that contains the total drift for all the predicted time points. Next, we need a diffusion array for each of the scenarios. The needed number of scenarios will be controlled using the scenario_size.

In [22]:
```
drift = (mu - 0.5 * sigma**2) * t
print("drift:\n", drift)
diffusion = {str(num_times): sigma * W[str(num_times)] for num_times in range(1, times + 1)}
print("diffusion:\n", diffusion)
```

```
drift:
 [-0.0002051  -0.0004102  -0.0006153  -0.0008204  -0.0010255  -0.0012306
 -0.0014357  -0.00164079 -0.00184589 -0.00205099 -0.00225609 -0.00246119
 -0.00266629 -0.00287139]
diffusion:
 {'1': array([ 0.00764166,  0.00381589, -0.00859894, -0.01708948, -0.01201354,
       -0.01645756, -0.01514635, -0.01010263, -0.0187712 , -0.03042938,
       -0.04517355, -0.04560998, -0.03066906, -0.03973627]), '2': array([-0.0040
6386, -0.01202027, -0.02254068, -0.02647356, -0.02181432,
       -0.01946962, -0.02966852, -0.04146411, -0.04476605, -0.05757272,
       -0.05929069, -0.05608349, -0.05731772, -0.0540528 ])}
```

Next we apply the above mentioned single step predictions expression. We will also add the initial stock price (So)to the array of predicted stock prices. This is done using the np.hstack. We add it because our predicted prices display information from 30th June.

In [23]:
```
S = np.array([So * np.exp(drift + diffusion[str(num_times)]) for num_times in range(1, times + 1)]) #final GBM equation  from previous discussions
S = np.hstack((np.array([[So] for num_times in range(times)]), S)) #adds So to the prediction series, since it's the starting point
print(S)
```

```
[[1966.12       1980.79568085 1972.82742109 1948.08691989 1931.22047886
  1940.65010436 1931.64873426 1933.78653099 1943.16605271 1925.99933063
  1903.28565484 1875.04454076 1873.84202342 1901.65908796 1884.10780175]
 [1966.12       1957.74460323 1941.83148526 1921.11564456 1913.18252213
  1921.72311059 1925.83924715 1905.90667028 1883.17117957 1876.57838211
  1852.31895506 1848.76024779 1854.3187443  1851.65168285 1857.32608153]]
```

The above are the two scenarios which the stock price can take. NOTE THAT, you are free to use as many scenarios as possible.

Next, we use the predicted stock prices are used to make a dataframa for the dates from 30th June to 20th July.

In [24]:
```
first_date = June_data["Date"].max()
last_date = dt.date( 2020, 7, 20 )
Dates=pd.bdate_range(start =first_date ,end = last_date)
Julydf=pd.DataFrame(S).T.set_index(Dates).reset_index(drop = False)
Julydf.columns=['Date','Prediction_1','Prediction_2']
Julydf
```

Out[24]:

	Date	Prediction_1	Prediction_2
0	2020-06-30	1966.120000	1966.120000
1	2020-07-01	1980.795681	1957.744603
2	2020-07-02	1972.827421	1941.831485
3	2020-07-03	1948.086920	1921.115645
4	2020-07-06	1931.220479	1913.182522
5	2020-07-07	1940.650104	1921.723111
6	2020-07-08	1931.648734	1925.839247
7	2020-07-09	1933.786531	1905.906670
8	2020-07-10	1943.166053	1883.171180
9	2020-07-13	1925.999331	1876.578382
10	2020-07-14	1903.285655	1852.318955
11	2020-07-15	1875.044541	1848.760248
12	2020-07-16	1873.842023	1854.318744
13	2020-07-17	1901.659088	1851.651683
14	2020-07-20	1884.107802	1857.326082

SECOND STAGE: Using July Data

Next, we extract the actual stock prices from 30th June to 20th July. This will be used in compiling the errors between the actual July stock price and the two predictions made.

In [25]:
```
starts = pd.to_datetime('2020-06-30')
ends= pd.to_datetime('2020-07-22')
pred_end_date = pd.to_datetime('2020-08-31')
NSE20['Date']= pd.to_datetime(NSE20['Date'],format='%y/%m/%d')
July=NSE20[NSE20['Date'].between(starts,ends)]
July['Date'] = July['Date'].astype('datetime64')
July_Data=July.sort_values(by='Date', ascending = False)
July_Data
```

```
C:\ProgramData\Anaconda\lib\site-packages\ipykernel_launcher.py:6: SettingWithCo
pyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stab
le/user_guide/indexing.html#returning-a-view-versus-a-copy
```

Out[25]:

	Date	Price	Open	High	Low	Vol.	Change %
86	2020-07-17	2236.81	2236.81	2236.81	2236.81	-	-0.0344

85	2020-07-16	2124.78	2124.78	2124.78	2124.78	-	-0.0501
84	2020-07-15	2057.96	2057.96	2057.96	2057.96	-	-0.0314
83	2020-07-14	2052.85	2052.85	2052.85	2052.85	-	-0.0025
82	2020-07-13	2048.87	2048.87	2048.87	2048.87	-	-0.0019
81	2020-07-10	2032.34	2032.34	2032.34	2032.34	-	-0.0081
80	2020-07-09	2025.31	2025.31	2025.31	2025.31	-	-0.0035
79	2020-07-08	1958.55	1958.55	1958.55	1958.55	-	-0.0330
78	2020-07-07	1887.17	1887.17	1887.17	1887.17	-	-0.0364
77	2020-07-06	1873.47	1873.47	1873.47	1873.47	-	-0.0073
76	2020-07-03	1891.30	1891.30	1891.30	1891.30	-	0.0095
75	2020-07-02	1917.67	1917.67	1917.67	1917.67	-	0.0139
74	2020-07-01	1930.75	1930.75	1930.75	1930.75	-	0.0068
73	2020-06-30	1966.12	1966.12	1966.12	1966.12	-	0.0183

We single out only the closing stock price data.

In [26]:
```
RealJuly=July_Data['Price']
Real=pd.DataFrame(RealJuly)
Real.columns=['Actual Price']
REAL=Real.sort_index(axis = 0)
REAL
```

Out[26]:

	Actual Price
73	1966.12
74	1930.75
75	1917.67
76	1891.30
77	1873.47
78	1887.17
79	1958.55
80	2025.31
81	2032.34
82	2048.87
83	2052.85
84	2057.96
85	2124.78
86	2236.81

In [27]:
```
JulyReset=REAL.reset_index(drop=True)
JulyReset.head()
```

Out[27]:

	Actual Price
0	1966.12

1	1930.75
2	1917.67
3	1891.30
4	1873.47

Next we combuine the predicted stock prices and the actual stock price into one dataframe.

In [28]:
```
df=pd.concat([JulyReset,Julydf],axis =1)
df.head()
```

Out[28]:

	Actual Price	Date	Prediction_1	Prediction_2
0	1966.12	2020-06-30	1966.120000	1966.120000
1	1930.75	2020-07-01	1980.795681	1957.744603
2	1917.67	2020-07-02	1972.827421	1941.831485
3	1891.30	2020-07-03	1948.086920	1921.115645
4	1873.47	2020-07-06	1931.220479	1913.182522

We calculate the error as the absolute value of the difference between the actual price and the predicted price.

In [29]:
```
df['Prediction_1 Error'] = abs(df['Actual Price'] - df['Prediction_1'])
df['Prediction_2 Error'] = abs(df['Actual Price'] - df['Prediction_2'])
df
```

Out[29]:

	Actual Price	Date	Prediction_1	Prediction_2	Prediction_1 Error	Prediction_2 Error
0	1966.12	2020-06-30	1966.120000	1966.120000	0.000000	0.000000
1	1930.75	2020-07-01	1980.795681	1957.744603	50.045681	26.994603
2	1917.67	2020-07-02	1972.827421	1941.831485	55.157421	24.161485
3	1891.30	2020-07-03	1948.086920	1921.115645	56.786920	29.815645
4	1873.47	2020-07-06	1931.220479	1913.182522	57.750479	39.712522
5	1887.17	2020-07-07	1940.650104	1921.723111	53.480104	34.553111
6	1958.55	2020-07-08	1931.648734	1925.839247	26.901266	32.710753
7	2025.31	2020-07-09	1933.786531	1905.906670	91.523469	119.403330
8	2032.34	2020-07-10	1943.166053	1883.171180	89.173947	149.168820
9	2048.87	2020-07-13	1925.999331	1876.578382	122.870669	172.291618
10	2052.85	2020-07-14	1903.285655	1852.318955	149.564345	200.531045
11	2057.96	2020-07-15	1875.044541	1848.760248	182.915459	209.199752
12	2124.78	2020-07-16	1873.842023	1854.318744	250.937977	270.461256
13	2236.81	2020-07-17	1901.659088	1851.651683	335.150912	385.158317
14	NaN	2020-07-20	1884.107802	1857.326082	NaN	NaN

In [30]:
```
df.drop(df.tail(1).index,inplace=True) # drop last 1 rows to remove the NaN when making plots
```

In [32]:
```
df.tail()
```

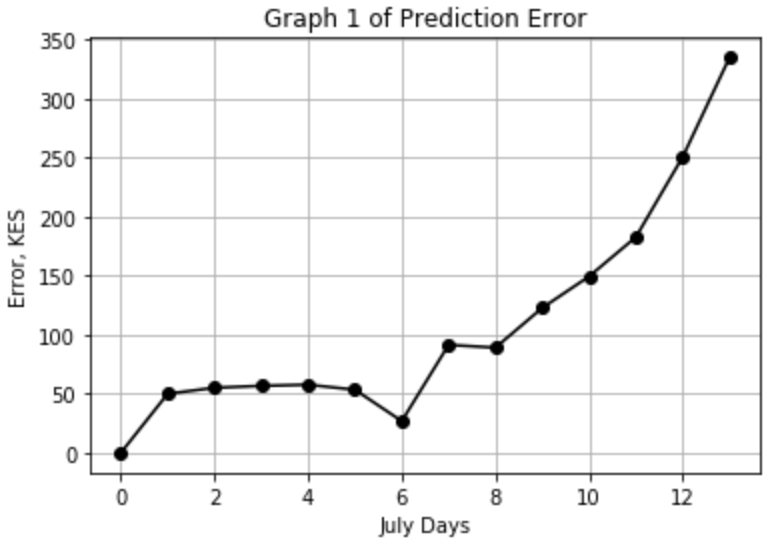
Out[32]:

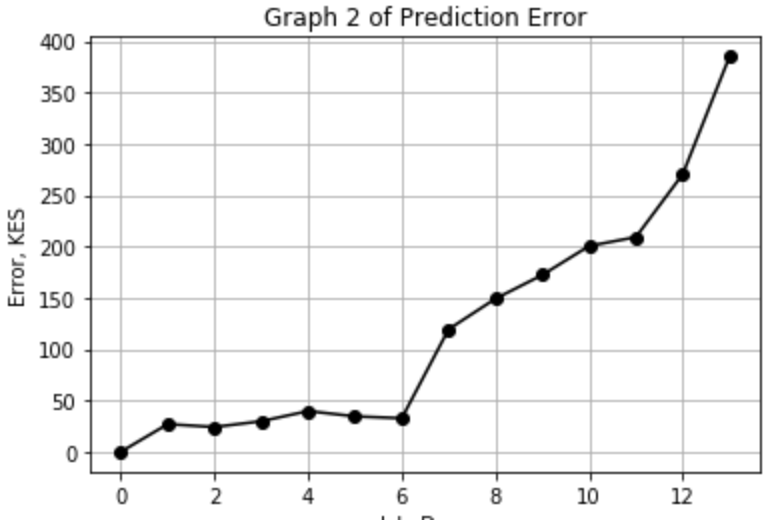
	Actual Price	Date	Prediction_1	Prediction_2	Prediction_1 Error	Prediction_2 Error
9	2048.87	2020-07-13	1925.999331	1876.578382	122.870669	172.291618
10	2052.85	2020-07-14	1903.285655	1852.318955	149.564345	200.531045
11	2057.96	2020-07-15	1875.044541	1848.760248	182.915459	209.199752
12	2124.78	2020-07-16	1873.842023	1854.318744	250.937977	270.461256
13	2236.81	2020-07-17	1901.659088	1851.651683	335.150912	385.158317

We make plots to visualize the error of the model.

In [33]:
```
df['Prediction_1 Error'].plot(marker='o',color='black')
plt.grid()
plt.xlabel('July Days')
plt.ylabel('Error, KES')
plt.title("Graph 1 of Prediction Error")
plt.show()

df['Prediction_2 Error'].plot(marker='o',color='black')
plt.grid()
plt.xlabel('July Days')
plt.ylabel('Error, KES')
plt.title("Graph 2 of Prediction Error")
plt.show()
```





FINAL STAGE: Making July - August 2020 Predictions (GBM)

Finally, after getting the idea of how the GBM works and the possible errors of the model, we move to the next core objective of the study of making July- August 2020 stock price predictions. We first retrieve the July data, compute mu and standard deviation. Thereafter compile the drift and diffusion components and apply the single time point expression.

In [34]:
```
START = pd.to_datetime('2020-07-01')
END = pd.to_datetime('2020-07-21')
NSE20['Date']= pd.to_datetime(NSE20['Date'],format='%y/%m/%d')
JULYDATA=NSE20[NSE20['Date'].between(START,END)]
JULYDATA['Date'] = JULYDATA['Date'].astype('datetime64')
JULYDATA=JULYDATA.sort_values(by='Date', ascending = False)
JULYDATA.head()
```

```
C:\ProgramData\Anaconda\lib\site-packages\ipykernel_launcher.py:5: SettingWithCo
pyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stab
le/user_guide/indexing.html#returning-a-view-versus-a-copy
  """
```

Out[34]:

	Date	Price	Open	High	Low	Vol.	Change %
86	2020-07-17	2236.81	2236.81	2236.81	2236.81	-	-0.0344
85	2020-07-16	2124.78	2124.78	2124.78	2124.78	-	-0.0501
84	2020-07-15	2057.96	2057.96	2057.96	2057.96	-	-0.0314
83	2020-07-14	2052.85	2052.85	2052.85	2052.85	-	-0.0025
82	2020-07-13	2048.87	2048.87	2048.87	2048.87	-	-0.0019

In [35]:
```
S_o = JULYDATA.loc[JULYDATA.shape[0]+73,"Price"]
print(S_o)
```

```
2236.81
```

In [36]:
```
JULYDATA.shape
```

Out[36]:
```
(13, 7)
```

In [37]:
```
DT=1
print(DT)
```

```
1
```

In [38]:
```
TT=np.busday_count('2020-07-22', '2020-08-11')
TT
```

Out[38]:
```
14
```

In [39]:
```
N = T/d_t
print(N)
```

```
14.0
```

In [40]:
```
t= np.arange(1, int(N) + 1)
print(t)
```

```
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14]
```

In [41]:
```
Returns = (JULYDATA['Price'].loc[85:] - JULYDATA['Price'].shift(1).loc[85:]) /JU
LYDATA['Price'].shift(1).loc[85:]
```

In [42]:
```
Mu = np.mean(Returns)
Sigma = np.mean(Returns)
```

In [43]:
```
times=2
b={str(num_times): np.random.normal(0, 1, int(N)) for num_times in range(1, time
s + 1)}
b
```

Out[43]:
```
{'1': array([-0.67650843,  0.45312398,  0.63780496,  0.20600253, -1.08885574,
        -1.54822693, -1.58590616,  0.58050895, -0.92237868, -1.07387448,
         1.45115657,  0.61657903,  0.19356338, -1.07021081]),
 '2': array([-0.95387979,  0.09552914, -1.73585727, -0.17259536, -0.39136056,
         1.18476824, -0.53330123, -0.82350665, -1.17237948, -1.46313314,
         1.9121082 , -0.50354747, -0.0455748 ,  0.44391842])}
```

In [44]:
```
W={str(num_times): b[str(num_times)].cumsum() for num_times in range(1, times +
1)}
W
```

Out[44]:
```
{'1': array([-0.67650843, -0.22338445,  0.4144205 ,  0.62042304, -0.4684327 ,
        -2.01665964, -3.6025658 , -3.02205685, -3.94443553, -5.01831001,
        -3.56715345, -2.95057442, -2.75701104, -3.82722185]),
 '2': array([-0.95387979, -0.85835065, -2.59420793, -2.76680329, -3.15816385,
        -1.97339561, -2.50669685, -3.3302035 , -4.50258298, -5.96571612,
        -4.05360793, -4.5571554 , -4.60273019, -4.15881177])}
```

In [45]:
```
Drift = (Mu - 0.5 * Sigma**2) * t
print("drift:\n", drift)
Diffusion = {str(num_times): Sigma * W[str(num_times)] for num_times in range(1,
times + 1)}
print("diffusion:\n", diffusion)
```

```
drift:
 [-0.0002051  -0.0004102  -0.0006153  -0.0008204  -0.0010255  -0.0012306
 -0.0014357  -0.00164079 -0.00184589 -0.00205099 -0.00225609 -0.00246119
 -0.00266629 -0.00287139]
diffusion:
 {'1': array([ 0.00764166,  0.00381589, -0.00859894, -0.01708948, -0.01201354,
        -0.01645756, -0.01514635, -0.01010263, -0.0187712 , -0.03042938,
        -0.04517355, -0.04560998, -0.03066906, -0.03973627]), '2': array([-0.0040
6386, -0.01202027, -0.02254068, -0.02647356, -0.02181432,
        -0.01946962, -0.02966852, -0.04146411, -0.04476605, -0.05757272,
        -0.05929069, -0.05608349, -0.05731772, -0.0540528 ])}
```

In [46]:
```
S = np.array([S_o * np.exp(Drift + Diffusion[str(num_times)]) for num_times in r
ange(1, times + 1)]) #final GBM equation  from previous discussions
```

```
S = np.hstack((np.array([[S_o] for num_times in range(times)]), S)) #adds So to
 the prediction series, since it's the starting point
print(S)
```

```
[[2236.81       2227.99153284 2189.35524383 2146.63073537 2115.66425472
  2117.76705601 2131.57918064 2146.45080226 2106.00973117 2103.89956983
  2105.61245303 2044.48820777 2005.10091945 1976.47083041 1977.99308474]
 [2236.81       2235.41286514 2206.08560724 2225.47434258 2203.3488874
  2187.17236765 2130.47382972 2118.43428584 2113.80452659 2118.02534384
  2129.66537493 2056.44675111 2044.09624426 2020.69446466 1985.87213831]]
```

Plotting Simulations

Having made the July - August 2020 NSE20 predictions, we make table and plot representations to visualize the predictions made.

In [48]:
```
first_date = dt.date( 2020, 7, 22)
last_date = dt.date( 2020, 8, 11)
Dates=pd.bdate_range(start =first_date ,end = last_date)
JULYDF=pd.DataFrame(S).T.set_index(Dates).reset_index(drop = False)
JULYDF.columns=['Date','First Prediction','Second Prediction']
JULYDF.head()
```

Out[48]:

	Date	First Prediction	Second Prediction
0	2020-07-22	2236.810000	2236.810000
1	2020-07-23	2227.991533	2235.412865
2	2020-07-24	2189.355244	2206.085607
3	2020-07-27	2146.630735	2225.474343
4	2020-07-28	2115.664255	2203.348887

In [49]:
```
JULYDF['First Prediction'].plot(marker='o',color='black')
plt.grid()
plt.xlabel('July-August Days')
plt.ylabel('NSE20 Price, KES')
plt.title("Graph of Predicted NSE20 Stock Price")
plt.show()

JULYDF['First Prediction'].plot(marker='o',color='black')
plt.grid()
plt.xlabel('17th July-21st August')
plt.ylabel('NSE20 Price, KES')
plt.title("Graph of Predicted NSE20 Stock Price")
plt.show()
```