FE-621 Homework 1

You Wang

03/17/2021

Part 1 In this problem, I use *yfinance* package to download data, code files are under **Part1** folder.

1. The follwing codes are from **getData.py**:

```
import yfinance as yf
import pandas as pd
from datetime import datetime


def getData(ticker):
    temp = yf.Ticker(ticker)
    expiry = temp.options
    data = []
    for date in expiry:
        op = temp.option_chain(date)
        op_data = op.calls.append(op.puts)
        op_data['date'] = datetime.today().date()
        op_data['spotPrice'] = temp.history().iloc[-1, :]["Close"]
        data.append(op_data)
        pd.to_pickle(data, "./data.pkl")
    return data
```

In the function, I concat the data of calls and puts into one *DataFrame*, then add a new column consisting of the **spot price** of the asset.

After downloading I save the data to a *pkl* file, which is a binary file supported by *pandas* package. An example of the data:

```
amzn1[0]
```

	contractSymbol	lastTradeDate	strike	lastPrice	bid	ask	change	percentChange	volume	openInterest	...	currency	date	spotl
0	AMZN210312C02000000	2021-02-08 16:34:21	2000.0	1310.00	1155.40	1168.10	0.00000	0.000000	NaN	1.0	...	USD	2021-02-24	3159.53
1	AMZN210312C02050000	2021-02-19 20:56:07	2050.0	1198.75	1105.50	1118.10	0.00000	0.000000	1.0	0.0	...	USD	2021-02-24	3159.53
2	AMZN210312C02100000	2021-02-24 14:56:36	2100.0	1044.40	1055.65	1069.00	-145.92993	-12.259621	2.0	1.0	...	USD	2021-02-24	3159.53
3	AMZN210312C02160000	2021-02-04 18:50:25	2160.0	1156.70	995.25	1010.05	0.00000	0.000000	NaN	1.0	...	USD	2021-02-24	3159.53
4	AMZN210312C02220000	2021-02-01 15:17:06	2220.0	1071.29	935.45	950.15	0.00000	0.000000	NaN	1.0	...	USD	2021-02-24	3159.53
...	...	...	...	...	...	...	...	...	...	...	...	...	...	
139	AMZN210312P03560000	2021-02-19 15:00:17	3560.0	256.35	396.65	409.75	0.00000	0.000000	1.0	1.0	...	USD	2021-02-24	3159.53
140	AMZN210312P03600000	2021-02-19 15:00:17	3600.0	292.60	434.30	449.05	0.00000	0.000000	1.0	2.0	...	USD	2021-02-24	3159.53
141	AMZN210312P03620000	2021-02-19 19:49:16	3620.0	439.35	455.70	468.75	0.00000	0.000000	1.0	1.0	...	USD	2021-02-24	3159.53
142	AMZN210312P03700000	2021-02-19 19:30:05	3700.0	427.90	535.00	548.00	0.00000	0.000000	3.0	3.0	...	USD	2021-02-24	3159.53
143	AMZN210312P03720000	2021-02-19 19:30:05	3720.0	447.40	553.15	567.85	0.00000	0.000000	3.0	3.0	...	USD	2021-02-24	3159.53

272 rows × 23 columns

Figure 1: image-20210226225709618

- Bonus:

  Here's the function to download multiple assets:

```
def getMultiData(tickers):
    for ticker in tickers:
        filepath = "./" + ticker + ".pkl"
        temp = yf.Ticker(ticker)
        expiry = temp.options
        data = []
        for date in expiry:
            op = temp.option_chain(date)
            op_data = op.calls.append(op.puts)
            op_data['date'] = datetime.today().date()
            op_data['spotPrice'] = temp.history().iloc[-1, :]["Close"]
            data.append(op_data)
            pd.to_pickle(data, filepath)
            pd.to_csv(data, filepath)
    return
```

  The only parameter of the function is tickers of the assets we want to download. It returns nothing, just save data to *pkl* files and *csv* files.

2. The following codes are from **cleanData.py**:

```
import yfinance as yf
import numpy as np
import pandas as pd

tickers = ['AMZN', 'SPY', '^VIX']
getMultiData(tickers)
# read data
amzn1 = pd.read_pickle("./amzn1.pkl")
amzn2 = pd.read_pickle("./amzn2.pkl")

spy1 = pd.read_pickle("./spy1.pkl")
spy2 = pd.read_pickle("./spy2.pkl")

vix1 = pd.read_pickle("./vix1.pkl")
vix2 = pd.read_pickle("./vix2.pkl")
```

The `getMultiData` function was done on two consecutive days because most APIs based on Yahoo Finance only offers us one day's option chain data. And the program uses `read_pickle` to read data after downloading.

```
# discard useless data
def clean_data(ticker, data):
    del data[0]
    temp = yf.Ticker(ticker)
    expiry = pd.Series(temp.options)
    expiry = pd.to_datetime(expiry)
    expiry = expiry - pd.Timestamp('today') < pd.Timedelta('80d')
    kept = len(expiry[expiry].index.values)
    newdata = data[0:kept]
    # add new columns of option type and expiry date
    def findType(s):
        if 'C' in s.replace(ticker, ''):
            return 'c'
        else:
            return 'p'
    def findDate(s):
        if 'VIXW' in s:
            return '20'+ s.replace('VIXW', '')[0:6]
        elif 'VIX' in s:
            return '20' + s.replace('VIX', '')[0:6]
        else:
            return '20' + s.replace(ticker, '')[0:6]
    for df in newdata:
        df['type'] = df['contractSymbol'].map(findType)
```

```
        df['expiry'] = df['contractSymbol'].map(findDate)
        df['expiry'] = pd.to_datetime(df['expiry'], infer_datetime_format=True)
        df['delta_t'] = (df['expiry'] - pd.Timestamp('today')) / np.timedelta64(
    return newdata


amzn1 = clean_data("AMZN", amzn1)
amzn2 = clean_data("AMZN", amzn2)

spy1 = clean_data("SPY", spy1)
spy2 = clean_data("SPY", spy2)

vix1 = clean_data("^VIX", vix1)
vix2 = clean_data("^VIX", vix2)

DATA1 = [amzn1, spy1, vix1]
DATA2 = [amzn2, spy2, vix2]
```

Afterwards I define a `clean_data` function:

- Only keep option chains which are maturing from 1 week to 80 days.
- From contract symbol grab option type and keep it in a new column **type**.
- From contract symbol get expiry date and keep it in a new column **expiry**.
- Calculate Time to Maturity and turn it to years, keep it in column **delta_t**.

Then use apply the function and save cleaned data to **DATA1** and **DATA2**.

3.
   - Take **AMZN210312C02220000** for example: In general, it means that an investor can **buy AMZN** at **2021-03-12** with price **$2200**. In the previous function, we used it to determine options' types and their expiration date.
     - **AMZN** is the ticker of the option's udnerlying asset.
     - **210312** stands for that it's maturying at **2021-03=12**.
     - **C** means it's a call option, For put, it would be **P**.
     - **02220000** means its strike is **$2200**.
   - **SPY** is the ticker of **SPDR S&P 500 Trust ETF**. The SPDR S&P 500 trust is an exchange-traded fund which trades on the NYSE Arca under the symbol. SPDR is an acronym for the Standard & Poor's Depositary Receipts,

the former name of the ETF. It is designed to track the S&P 500 stock market index.

- **VIX** is the ticker symbol for the Chicago Board Options Exchange's CBOE **Volatility Index**, a popular measure of the stock market's expectation of volatility based on S&P 500 index options. It is calculated and disseminated on a real-time basis by the CBOE, and is often referred to as the fear index or fear gauge.

4.
   - The underlying equity and ETF price has been downloaded as downloading option chains and has been added to the *DataFrame*.
   - I also use 3-month Treasury bills as short-term interest rate, which is 0.03% for Feb 24 and 0.04% for Feb 25.
   - Time to Maturity is also being calculated in the data-cleaning process, which is talked in *Problem 2*.

Part 2 Codes used for problems in Part 2 are under **Part2** folder.

5. Black-Scholes formula is given by:

$$C = N(d_1) S_t - N(d_2) K e^{-rt}$$
$$\text{where } d_1 = \frac{\ln \frac{S_t}{K} + \left(r + \frac{\sigma^2}{2}\right) t}{\sigma\sqrt{t}}$$
$$\text{and } d_2 = d_1 - \sigma\sqrt{t}$$

   The following codes are from **BS_formula.py**:

```
import numpy as np
from scipy.stats import norm


def BS_formula(Type, S, K, T, sigma, r):
    d1 = (np.log(S / K) + (r + sigma ** 2 / 2) * T) / (sigma * np.sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)
    if Type == 'c':
        return norm.cdf(d1) * S - norm.cdf(d2) * K * np.exp(-r * T)
    elif Type == 'p':
        return K * np.exp(-r * T) * norm.cdf(-d2) - norm.cdf(-d1) * S
    else:
        raise TypeError("Type must be 'c' for call, 'p' for put")
```

6. Code of Bisection method is from **root_method.py**:

```
def bisection(f, a, b, tol=1e-6):
    if f(a) == 0:
        return a
    elif f(b) == 0:
        return b
    while abs(a - b) >= tol:
        c = (a + b) / 2
        if f(c) == 0:
            break
        if f(a) * f(c) < 0:
            b = c
        else:
            a = c
    return c
```

Then we can define a function used to calculate implied volatility:

```
def get_impliedVol(Type, S, K, T, r, P, method):
    """
    To calculate the implied vol using specified method

    :param Type: type of the option
    :param S: spot price
    :param K: strike
    :param T: time to maturity
    :param r: risk-free interest rate
    :param P: price of the option
    :param method: the method to find root, points to the function of the meth
    :return:
    """
    def price_diff(sigma):
        return BS_formula(Type, S, K, T, sigma, r) - P
    if method == 'b':
        return bisection(price_diff, 0.001, 1)
    elif method == 'n':
        price_diff_prime = lambda x: vega(S, K, T, x, r)
        return newton_method(price_diff, price_diff_prime, 0.5)
```

At last use `apply` method in *pandas* package to apply the function to each axis of the DataFrame and record its running time:

```
# calculate implied vol
r = 0.07/100
now = pd.Timestamp.now()
```

```
for i in range(2):
    for df in DATA1[i]:
        df['optionPrice'] = df['bid']/2 + df['ask']/2
        df['bisec_vol'] = df.apply(lambda x: get_impliedVol(x.type, x.spotPrice,
end = pd.Timestamp.now()
bisec_time = (end - now) / np.timedelta64(1, 's')
print("It takes {} seconds".format(bisec_time))
```

```
r = 0.07/100
now = pd.Timestamp.now()
for i in range(2):
    for df in DATA1[i]:
        df['bisec_Root'] = df.apply(lambda x: get_impliedVol(x.type, x.spotPrice, x.strike,
                                                             x.delta_t, r, x.optionPrice, 'b'), axis=1)
end = pd.Timestamp.now()
bisec_time = (end - now) / np.timedelta64(1, 's')
print("It takes {} seconds".format(bisec_time))
```

```
It takes 45.684028 seconds
```

Figure 2: image-20210227160239730

Next, use the following funciton to determine option's moneyness:

```
def moneyness(S, K):
    if S/K < 0.95:
        return 'inTheMoney'
    elif S/K > 1.05:
        return 'outTheMoney'
    else:
        return 'atTheMoney'

# moneyness
for i in range(3):
    for df in DATA1[i]:
        df['moneyness'] = df.apply(lambda x: moneyness(x.spotPrice, x.strike), a
```

Here are one of the tables of options **at-the-money**:

- For options with underlying AMZN matures at 2021-05-21:
- For options with underlying SPY matures at 2021-03-19:

At last, for each option between in-the-money and out-of-the-money, calculate the average of them:

```
# amzn
expiry_date = []
in_mean = []
out_mean = []
at_mean = []
for df in DATA1[0]:
    expiry_date.append(df.expiry.iloc[0])
    temp = df[df.moneyness == 'atTheMoney']['bisec_Root'].mean()
    at_mean.append(temp)
    temp = df[df.moneyness == 'inTheMoney']['bisec_Root'].mean()
    in_mean.append(temp)
    temp = df[df.moneyness == 'outTheMoney']['bisec_Root'].mean()
    out_mean.append(temp)
amzn_mean = pd.DataFrame({'expiry': expiry_date, 'at-the-money': at_mean,
                          'out-of-the-money': out_mean, 'in-the-money': in_mean})
# spy
expiry_date = []
in_mean = []
out_mean = []
at_mean = []
for df in DATA1[1]:
    expiry_date.append(df.expiry.iloc[0])
    temp = df[df.moneyness == 'atTheMoney']['bisec_Root'].mean()
    at_mean.append(temp)
    temp = df[df.moneyness == 'inTheMoney']['bisec_Root'].mean()
    in_mean.append(temp)
    temp = df[df.moneyness == 'outTheMoney']['bisec_Root'].mean()
    out_mean.append(temp)
spy_mean = pd.DataFrame({'expiry': expiry_date, 'at-the-money': at_mean,
                          'out-of-the-money': out_mean, 'in-the-money': in_mean})
```

Here's the table of average implied volatilities:

7. To use **Newton method**, we need to calulate the option's derivative with respect to the volatility σ first, which is **Vega**:

$$Vega = \frac{\partial V}{\partial \sigma} = SN'(d_1)\sqrt{T-t}$$

```
def vega(S, K, T, sigma, r):
    d1 = (np.log(S/K) + (r+0.5*sigma**2)*T)/(sigma*np.sqrt(T))
    return np.sqrt(T)*S*norm.pdf(d1)
```

Next define the Newton method:

```
def newton_method(f, f_prime, x0, tol=1e-6, N=1000):
    for i in range(N):
        x1 = x0 - f(x0)/f_prime(x0)
        if abs(x1- x0) < tol:
            break
        x0 = x1
    return x1
```

Set the `getimpliedVol` function, which is mentioned before:

```
def get_impliedVol(Type, S, K, T, r, P, method):
    def price_diff(sigma):
        return BS_formula(Type, S, K, T, sigma, r) - P
    if method == 'b':
        return bisection(price_diff, 0.001, 1)
    elif method == 'n':
        price_diff_prime = lambda x: vega(S, K, T, x, r)
        return newton_method(price_diff, price_diff_prime, 0.5)
```

And then we use the same method to calculate implied vol and their average:

```
now = pd.Timestamp.now()
for i in range(2):
    for df in DATA1[i]:
        df['newton_Root'] = df.apply(lambda x: get_impliedVol(x.type, x.spotPric
                                                              x.delta_t, r, x.opt
end = pd.Timestamp.now()
newton_time = (end - now) / np.timedelta64(1, 's')
print("It takes {} seconds".format(newton_time))
```

```
now = pd.Timestamp.now()
for i in range(2):
    for df in DATA1[i]:
        df['newton_Root'] = df.apply(lambda x: get_impliedVol(x.type, x.spotPrice, x.strike,
                                                              x.delta_t, r, x.optionPrice, 'n'), axis=1)
end = pd.Timestamp.now()
newton_time = (end - now) / np.timedelta64(1, 's')
print("It takes {} seconds".format(newton_time))
```

```
<ipython-input-4-f49d1676a7fe>:31: RuntimeWarning: divide by zero encountered in double_scalars
  x1 = x0 - f(x0)/f_prime(x0)
<ipython-input-4-f49d1676a7fe>:14: RuntimeWarning: invalid value encountered in double_scalars
  d1 = (np.log(S / K) + (r + sigma ** 2 / 2) * T) / (sigma * np.sqrt(T))
<ipython-input-4-f49d1676a7fe>:25: RuntimeWarning: invalid value encountered in double_scalars
  d1 = (np.log(S/K) + (r+0.5*sigma**2)*T)/(sigma*np.sqrt(T))
<ipython-input-4-f49d1676a7fe>:14: RuntimeWarning: overflow encountered in double_scalars
  d1 = (np.log(S / K) + (r + sigma ** 2 / 2) * T) / (sigma * np.sqrt(T))
<ipython-input-4-f49d1676a7fe>:25: RuntimeWarning: overflow encountered in double_scalars
  d1 = (np.log(S/K) + (r+0.5*sigma**2)*T)/(sigma*np.sqrt(T))
<ipython-input-4-f49d1676a7fe>:31: RuntimeWarning: overflow encountered in double_scalars
  x1 = x0 - f(x0)/f_prime(x0)
```

```
It takes 16.507666 seconds
```

Figure 3: image-20210227171851521

Note that **Newton's method** does not always converge, Its convergence theory is for "local" convergence which means we should start close to the root, where "close" is relative to the function we're dealing with. So sometimes when options are mispriced, we can't get convergence and our program will reveal warning.

Also, **Newton's method** is the fastest method compared with other algorithms, but it needs us to have the formula of its derivative. If not, we have to use **Secant method**, which would be slower.

The following is a table containing implied vols calculated by both bisection method and Newton's method:

Obviously their results are very close to each other, but the running time of **Newton's method** is less:

8. I have output all the data of opitons to csv files in **'./Patr2/files'**, files' names are like **AMZNexp2021-04-01**, means the file contains all the AMZN option data maturing at 2021-04-01.

```
info = ['contractSymbol', 'expiry', 'type', 'optionPrice', 'bisec_Root', 'newton
tickers = ['AMZN', 'SPY']
 for i in range(2):
     for df in DATA1[i]:
         filepath = './files/' + tickers[i] + 'exp' + str(df.expiry.iloc[0].date
         df.to_csv(filepath)
```

and here's an example of **AMZNexp2021-04-01.csv**:

To calculate the average volatilities for every maturity, type, stock/ETF:

```
# AMZN
expiry_date = []
call_vol = []
put_vol = []
for df in DATA1[0]:
    expiry_date.append(df.expiry.iloc[0])
    temp = df[df.type=='c']['bisec_Root'].mean()
    call_vol.append(temp)
    temp = df[df.type=='p']['bisec_Root'].mean()
    put_vol.append(temp)
amzn_vol = pd.DataFrame({'expiry': expiry_date, 'vol of call': call_vol,
                         'vol of put': put_vol})
# spy
expiry_date = []
call_vol = []
put_vol = []
for df in DATA1[1]:
    expiry_date.append(df.expiry.iloc[0])
    temp = df[df.type=='c']['bisec_Root'].mean()
    call_vol.append(temp)
    temp = df[df.type=='p']['bisec_Root'].mean()
    put_vol.append(temp)
spy_vol = pd.DataFrame({'expiry': expiry_date, 'vol of call': call_vol,
                         'vol of put': put_vol})

# concat above tables
vol_table = pd.concat([amzn_vol, spy_vol], keys=['AMZN', 'SPY'])
```

The following is the volatility table:

Comment:

- Within 80 days, the number of SPY options are bigger than AMZN options, which implies that investors have a stronger short-term demand of SPY options.
- With the same expiration data, implied volatilities of AMZN options are larger than SPY options. It's appropriate because SPY ETF comprises 500 large- and mid-cap U.S. stocks.

Comparison to the price of VIX:

The price of **VIX** on the first day is 21.34. By the definition of **VIX**, it means

that SPY's implied volatility is 21.34%.

- In general, **SPY optons'** implied volatilities are very close to VIX price. For **AMZN options**, their implied volatilitie are larger than 21.34%.
- By taking a look at a slice of volatility table:

  We can find that the further out of the money the option is, the larger its implied volatility and futher to the the price of **VIX**.

9. The **Put-Call Parity** is defined as:

$$C - P = S - Ke^{-rT}$$

Write it into program:

```
def parity(Type, S, K, T, r, P):
    if Type == 'c':
        return P - S + K * np.exp(-r*T)
    elif Type == 'p':
        return P + S - K * np.exp(-r*T)
```

Then apply it into DataFrames:

```
for i in range(2):
    for df in DATA1[i]:
        df['parity_price'] = df.apply(lambda x: parity(x.type, x.spotPrice,
                                                       x.strik
                                      axis=1)
```

Then I choose **SPY Options** matruing at 2021-03-22, deal with the format of the data to make it more readable:

```
# choose data with the same strike
temp = DATA1[1][8]
dup = temp['strike'].duplicated(keep=False)
temp = temp[dup]
temp = temp.sort_values(by='strike')
info = ['strike', 'bid', 'ask', 'parity_price']

# merge call and put dataframe
temp_p = temp[temp.type=='p'][info]
temp_c = temp[temp.type=='c'][info]
temp = pd.merge(temp_p, temp_c, on='strike')
temp = temp.rename(columns={'bid_x':'bid of put', 'ask_x':'ask of put',
                            'bid_y':'bid of call', 'ask_y':'ask of call', 'parit
```

It's hard to compare the prices calculated by **Put-Call Parity** with BID/ASK prices directly, we consider to calculate the average difference of them:

```
bp_diff = (temp['bid of put'] - temp['parity price by call']).mean()
ap_diff = (temp['ask of put'] - temp['parity price by call']).mean()
bc_diff = (temp['bid of call'] - temp['parity price by put']).mean()
ac_diff = (temp['bid of call'] - temp['parity price by put']).mean()
```

As we can see here, for the data we chose before:

- Put prices calculated by **Put-Call Parity** is more close to bid prices.
- Put prices calculated by **Put-Call Parity** is more close to ask prices.

10. Here I choose SPY call options' volatilities maturing at 2021-03-15, 2021-03-22, 2021-03-26 to plot:

```
temp1 = DATA1[1][5]
temp2 = DATA1[1][8]
temp3 = DATA1[1][10]
temp1 = temp1[temp1.type == 'c']
temp2 = temp2[temp2.type == 'c']
temp3 = temp3[temp3.type == 'c']

exp = [temp1.expiry.iloc[0], temp2.expiry.iloc[0], temp3.expiry.iloc[0]]
strike1 = temp1.strike
strike2 = temp2.strike
strike3 = temp3.strike
vol1 = temp1.bisec_Root
vol2 = temp2.bisec_Root
vol3 = temp3.bisec_Root

plt.figure(1)
plt.plot(strike1, vol1, label=exp[0].date())
plt.plot(strike2, vol2, label=exp[1].date())
plt.plot(strike3, vol3, label=exp[2].date())
plt.xlabel('strike')
plt.ylabel('vol')
plt.title("SPY call options volatility")
plt.legend()
```

We can observe the volatilty smile: these implied volatilities can create a line that slopes upward on either end.

- Bonus:

```
import matplotlib as mpl
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm

# merge strike and vol of call options
info = ['strike', 'bisec_Root']
delta_T = ['strike', 1]
new_df = DATA1[1][0][info]
for df in DATA1[1]:
    delta_T.append(df['delta_t'].iloc[0]*365)
    call_df = df[df.type == 'c']
    new_df = pd.merge(new_df, call_df[info], on='strike', how='outer')
    new_df.columns = delta_T

# drop useless data
new_df = new_df.set_index('strike')
new_df = new_df.drop(new_df.columns[0:4], axis=1)
new_df = new_df.dropna()

# 3d plot
x = new_df.columns
y = new_df.index
X,Y = np.meshgrid(x,y)
Z = new_df
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
surf = ax.plot_surface(Y,X,Z, cmap=cm.coolwarm,
                       linewidth=0, antialiased=False)
ax.set_title('Implied volatility surface')
ax.set_ylabel('time to maturity')
ax.set_xlabel('strike')
ax.set_zlabel('volatility')
plt.show()
```

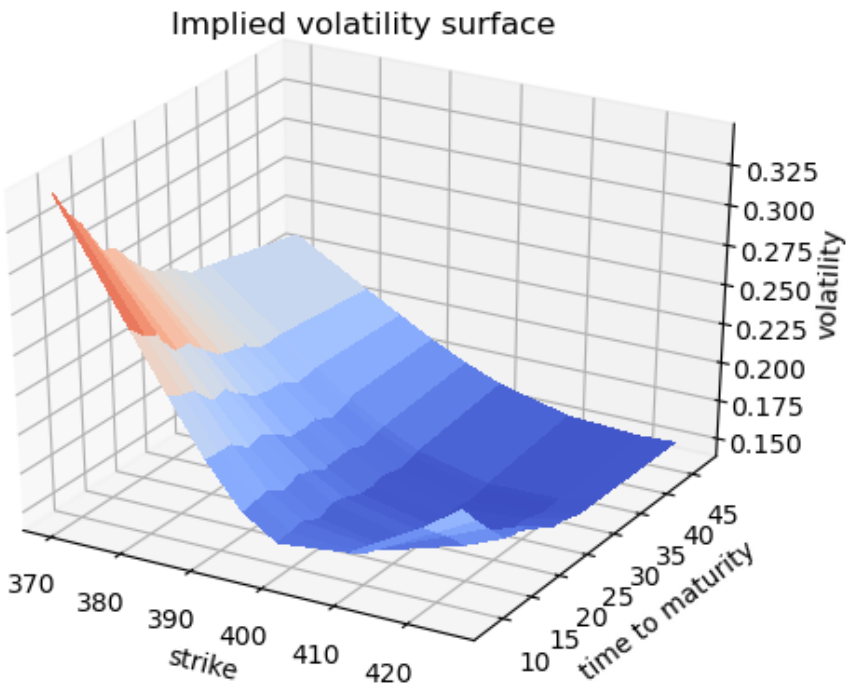

Figure 4: image-20210228022435562

11. First we define greeks functions:

```
def delta(Type, S, K, T, r, sigma):
    d1 = (np.log(S / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
    if Type == 'c':
        return norm.cdf(d1)
    elif Type == 'd':
        return norm.cdf(d1) - 1


def gamma(S, K, T, r, sigma):
    d1 = (np.log(S / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np.sqrt(T))
    return norm.pdf(d1) / (S * sigma * np.sqrt(T))


def vega(S, K, T, sigma, r):
```

```
    d1 = (np.log(S/K) + (r+0.5*sigma**2)*T)/(sigma*np.sqrt(T))
    return np.sqrt(T)*S*norm.pdf(d1)
```

Then we define funcitons used for approximation:

```
def prime(f, x, h=1e-4):
    return (f(x+h) - f(x)) / h

def dprime(f, x, h=1e-4):
    return (prime(f,x+h) - prime(f,x)) / h

def delta_approx(S, K, T, r, sigma, h=1e-4):
    s = lambda x: BS_formula('c', x, K, T, sigma, r)
    return prime(s, S, h)

def gamma_approx(S, K, T, r, sigma, h=1e-4):
    s = lambda x: BS_formula('c', x, K, T, sigma, r)
    return dprime(s, S, h)

def vega_approx(S, K, T, r, sigma, h=1e-4):
    sig = lambda x: BS_formula('c', S, K, T, x, r)
    return prime(sig, sigma, h)
```

Then apply these function:

```
for i in range(2):
    for df in DATA1[i]:
        df['delta'] = df.apply(lambda x:
                               delta(x.type, x.spotPrice, x.strike,
                                     x.delta_t, r, x.bisec_Root),
                               axis=1)
        df['delta_approx'] = df.apply(lambda x:
                                      delta_approx(x.spotPrice, x.strike,
                                            x.delta_t, r, x.bisec_Root),
                                      axis=1)
        df['gamma'] = df.apply(lambda x:
                               gamma(x.spotPrice, x.strike, x.delta_t,
                                     r, x.bisec_Root),
                               axis=1)
        df['gamma_approx'] = df.apply(lambda x:
                                      gamma_approx(x.spotPrice, x.strike,
                                           x.delta_t, r, x.bisec_Root),
                                      axis=1)
        df['vega'] = df.apply(lambda x:
```

```
                                vega(x.spotPrice, x.strike, x.delta_t,
                                    x.bisec_Root, r), axis=1)
        df['vega_approx'] = df.apply(lambda x:
                                        vega_approx(x.spotPrice, x.strike, x.delta_t
                                                    r, x.bisec_Root), axis=1)
```

Choose SPY call options maturing at 2021-03-15 to output table of Greeks:

```
greeks_info = ['contractSymbol', 'delta', 'delta_approx',
               'gamma', 'gamma_approx', 'vega', 'vega_approx']
temp = DATA1[1][9]
temp = temp[temp.type=='c']
temp[greeks_info]
```

Greeks calculated by these two methods are very close to each other.

12. It's easy to use `apply` function to calculate the price using volatility calculated in DATA1:

```
for i in range(2):
    for j in range(len(DATA1[i])):
        DATA2[i][j]['vol'] = DATA1[i][j].bisec_Root
        DATA2[i][j]['BS_price'] = DATA2[i][j].apply(lambda x:
                                                    BS_formula(x.type, x.spotPric
                                                               x.strike, x.delta_t
                                                               x.vol, r), axis=1)
```

We can find that prices calculated by using implied volatility from DATA1 are very close to the average of bid and ask prices.

Part 3 Codes used to solve Part 3 problems are under **Part 3** folder.

Define $f(x)$:

```
import numpy as np
import matplotlib.pyplot as plt


def f(x):
    if x == 0:
        return 1
    else:
        return np.sin(x) / x
```

1.
```
def trapezoidal(f, a, b, N=1000000):
    x = np.linspace(a, b, N + 1)
    g = np.vectorize(f)
    return (b - a) / N * (np.sum(g(x)) - f(a) / 2 - f(b) / 2)


def simpson(f, a, b, N=1000000):
    x = np.linspace(a, b, N + 1)
    g = np.vectorize(f)
    h = (b - a) / N
    return h / 3 * (g(a) + np.sum(g(x[1:-1:2])) * 4
                    + np.sum(g(x[2:-1:2])) * 2 + g(b))
```

2. First define truncation and difference functions:

```
def trap_trunc(a=1e5, N=100000):
    return abs(trapezoidal(f, -a, a, N) - np.pi)


def simp_trunc(a=1e5, N=100000):
    return abs(simpson(f, -a, a, N) - np.pi)


def diff(a=1e5, N=100000):
    return trapezoidal(f, -a, a, N) - simpson(f, -a, a, N)
```

Next we study the differences from two angles:

- Fix a, study changes when N increases
- Fix N, study changes when a increases

```
# a is fixed, N increases
N = np.arange(50, 10000, 50)
error_t1 = [trap_trunc(N=i) for i in N]
error_s1 = [simp_trunc(N=i) for i in N]
diff1 = [diff(N=i) for i in N]

# N is fixed, a increases
a = np.arange(50, 10000, 50)
error_t2 = [trap_trunc(a=i) for i in a]
error_s2 = [simp_trunc(a=i) for i in a]
diff2 = [diff(a=i) for i in a]
```

```
t1 = plt.subplot(2, 2, 1)
plt.plot(N,error_t1)
t2 = plt.subplot(2, 2, 2)
plt.plot(a, error_t2)
s1 = plt.subplot(2, 2, 3)
plt.plot(N, error_s1)
s2 = plt.subplot(2, 2, 4)
plt.plot(N, error_s2)
plt.show()

t1 = plt.subplot(2, 1, 1)
plt.plot(N, diff1)
t2 = plt.subplot(2, 1, 2)
plt.plot(a, diff2)
plt.show()
```

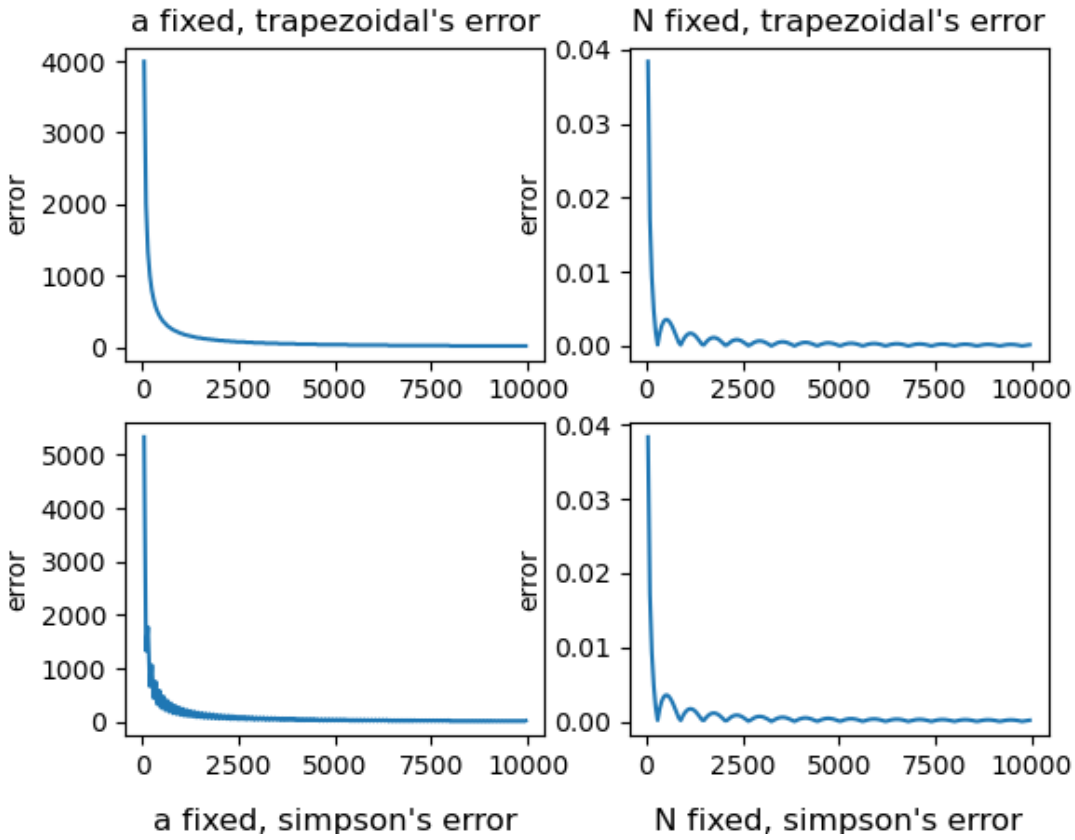

Figure 5: image-20210227230614519

We can find that:

- With a increases, the error of both algorithms converge to 0 quickly, and there are small fluctuations in simpson's error.
- With N increases, the error of both algorithms decreases with fluctuaions.

Now we look at the difference between the two algorithms:

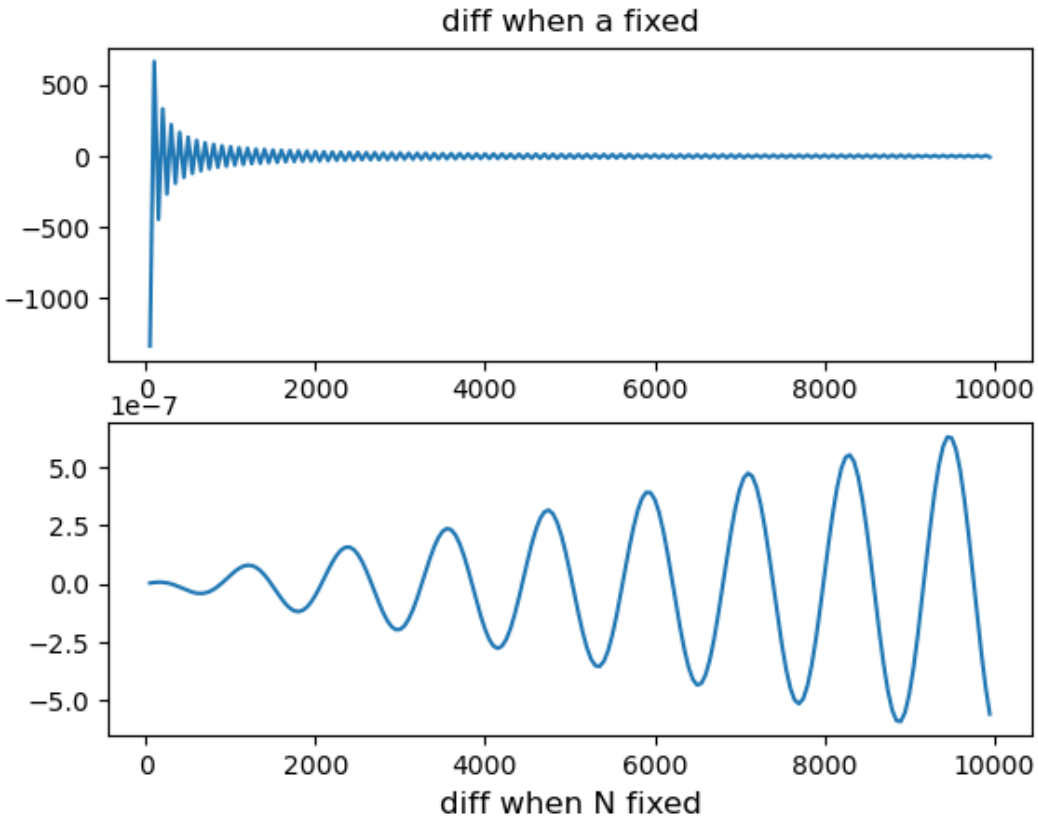

Figure 6: image-20210227231758047

- When a is fixed, the difference between two algorithms decreases quickly.
- When N is fixed, with a increases, the difference between two algorithms fluctuates, and the amplitude becomes larger and larger. I think it's because when a increases, trapezoidal method truncates more value and Simpson's rules become more precise than trapezoidal.

3. I follow the problem to write the `integral` function:

```
def integral(f, a, b, tol=1e-4, rule=trapezoidal):
    step = 10
    I_k = rule(f, a, b, step)
    I_k1 = rule(f, a, b, step+1)
    while abs(I_k - I_k1) >= tol:
        step += 1
        I_k = I_k1
        I_k1 = rule(f, a, b, step+1)
```

```
    return I_k1, step
```

Use it to integrate $f(x)$:

```
res1, step1 = integral(f, -1e4, 1e4)
res2, step2 = integral(f, -1e4, 1e4, rule=simpson)
print("The value using trapezoidal is {0}, and take {1} steps".format(res1, step
print("The value using simpson is {0}, and take {1} steps".format(res2, step2))
```

As we can see, Simpson's rule takes more steps and its result is more precise.

4. Define inline function g and integrate it from 0 to 2:

```
g = lambda x: 1 + np.exp(-x**2)*np.sin(8*x**(2/3))

res3, step3 = integral(g, 0, 2)
res4, step4 = integral(g, 0, 2, rule=simpson)
print("The value using trapezoidal is {0}, and take {1} steps".format(res3, step
print("The value using simpson is {0}, and take {1} steps".format(res4, step4))
```

And we can find that their results are very similar but simpson's rule take much longer time to finish calculating.