Optimizing Trading Schedules Under Memory Based Liquidity Impact

In this project we make use of dynamic programming to create a trading schedule that maximizes total number of shares traded, under a model of liquidity impact with memory

Suppose we have a total of N shares that we would like to trade over T time periods. To do so, we produce a schedule

$$(n_0, n_1, \ldots, n_{T-1}) \quad \text{where each} \quad n_i \geq 0$$

Each n_i represents the quantity that we will attempt to trade at time $i = 0, 2, \ldots, T-1$

In reality the market will only allow us to trade a smaller quantity at each time period. We impose the following conditions:

$$\sum_{i=0}^{T-2} n_i \ \leq N \quad \text{and} \quad n_{T-1} = N - \text{quantity traded so far}$$

#

This plays out as follows. Assume that $\alpha > 0$ (and very small) and $0 < \pi < 1$ are given parameters. Then we run the following process:

1. Initialize $M = 0$. Then for $i = 0, 2, \ldots, T-1$ we do the following:

2. Compute:

$$M \leftarrow \lceil 0.1 * M + 0.9 * n_i \rceil$$

3. At time $i \leq T-1$ we trade,

$$S_i \ = \ \lceil (1 - \alpha M^{\pi}) n_i \rceil \ shares$$

4. Note that:

$$n_{T-1} = N \ - \sum_{i=0}^{T-2} n_i$$

Task 1

Code a dynamic programming algorithm that computes an optimal schedule of trades

$$(n_0, n_1, \ldots, n_{T-1})$$

The goal is to maximize the total number of traded shares: $\sum_{i=0}^{T-1} S_i$

Make sure that your code runs well for a range of values of α and π

Compute the optimal schedule when $\alpha = 0.001$, $\pi = 0.5$, $N = 100000$ and $T = 10$. Denote this schedule by $(S_0, S_1, \ldots, S_9)$.

In [1]:
```
import numpy as np
import pandas as pd
from tqdm.notebook import tqdm
```

In [2]:
```
# Outputs the optimal trading schedule for given alpha, T (time periods)
def trade_schedule(alpha: float, T: int, N: int, pi: float) -> list:
    '''
        alpha --> market memory impact factor 1
        pi    --> market memory impact factor 2
        T     --> total number of time periods
        N     --> total number of shares you have
    '''

    '''
        Computes optimal trade schedule for the given market impact model
        M(t) = ceil[ 0.1*M(t-1) + 0.9*N(t) ]
        S(t) = ceil[ (1 - alpha*M(t)^pi) * N(t) ]
    '''

    N = int(N)

    mem_S = np.zeros((T+1, N+1, N+1)).astype(int) # Memoization array to
    mem_N = np.zeros((T+1, N+1, N+1)).astype(int) # Memoization array to

    shares = np.arange(N+1).astype(int) # Creates array [0, 1, 2, ... N]
    reversed_shares = N - shares # Reversed the shares array
    reshaped_shares = shares.reshape(-1, 1) # Converts shares array to a

    impact = 1 - (alpha * (shares ** pi)) # Pre-computes the correspondin


    for t in range(T-1, -1, -1):
        # print("t =", T-t)
        for n in range(0, N+1, 1):

            # temp_m is a Nxn array which generates all possible values o
            # used to get all possible impacts in the next step for temp_
```

```
                temp_m = np.ceil( 0.1*reshaped_shares + 0.9*shares[ : n+1] ).

                temp_m[temp_m > N] = N # Needed... because ceil might cause a

                # temp_sell generates all possible shares we could be allowed
                temp_sell = np.ceil( impact[temp_m] * shares[ : n+1 ] ) + mem

                # Get the indices of max shares it's possible to sell for all
                idx_max = np.argmax( temp_sell, axis=1 )

                # Stores all possible max shares sold at current time period
                mem_S[t, : , n] = temp_sell[shares, idx_max]

                # Stores the index corresponding to max shares sold at time p
                mem_N[t, : , n] = idx_max

                # Reset
                temp_m = temp_sell = idx_max = None

    mem_S = shares = reversed_shares = reshaped_shares = impact = None

    schedule = list()
    m = 0
    remaining = N

    # Getting the optimal trading schedule back from mem_N
    for t in range(T):
        nt = mem_N[t, m, remaining]
        schedule.append(nt)

        m = np.ceil( 0.1*m + 0.9*nt ).astype(int)
        remaining -= nt

    return schedule
```

In [3]:
```
test_inputs = [
    {"alpha": 1e-2, "T": 4, "N": 1000, "pi": 0.3},
    {"alpha": 1e-2, "T": 3, "N": 1000, "pi": 0.3},
    {"alpha": 1e-1, "T": 4, "N": 500, "pi": 0.2},
    {"alpha": 1e-5, "T": 4, "N": 500, "pi": 0.2},
    {"alpha": 1e-5, "T": 4, "N": 500, "pi": 0.5},
]

tqdm_test_inputs = tqdm(test_inputs, leave=True)

for input in tqdm_test_inputs:
    print("Optimal trade schedule for: ", input)
    print( trade_schedule(**input) )
```

```
  0%|          | 0/5 [00:00<?, ?it/s]
```

```
Optimal trade schedule for:  {'alpha': 0.01, 'T': 4, 'N': 1000, 'pi': 0.
3}
[122, 281, 277, 320]
Optimal trade schedule for:  {'alpha': 0.01, 'T': 3, 'N': 1000, 'pi': 0.
3}
[239, 349, 412]
Optimal trade schedule for:  {'alpha': 0.1, 'T': 4, 'N': 500, 'pi': 0.2}
[101, 126, 135, 138]
Optimal trade schedule for:  {'alpha': 1e-05, 'T': 4, 'N': 500, 'pi': 0.
2}
[0, 0, 0, 500]
Optimal trade schedule for:  {'alpha': 1e-05, 'T': 4, 'N': 500, 'pi': 0.
5}
[0, 0, 0, 500]
```

In [4]:
```
# If the number of shares held - N - is too large, we can divide N by a p
# We make a small adjustment to the alpha to approximate the new impact
# Finally, we multiply the trade schedule with the factor (power of 10)
def approx_trade_schedule(alpha: float, T: int, N: int, pi: float, approx
    new_alpha = alpha * (10**(approx_factor*pi))
    new_N = int(N / (10**approx_factor))

    schedule = trade_schedule(new_alpha, T, new_N, pi)
    schedule = [ ni*(10**approx_factor)  for ni in schedule]

    return schedule
```

In [5]:
```
test_inputs = [
    {"alpha": 1e-2, "T": 4, "N": 1000, "pi": 0.3, "approx_factor": 1},
    {"alpha": 1e-2, "T": 3, "N": 1000, "pi": 0.3, "approx_factor": 1},
    {"alpha": 1e-1, "T": 3, "N": 2000, "pi": 0.1, "approx_factor": 1},
    {"alpha": 1e-5, "T": 4, "N": 500, "pi": 0.5, "approx_factor": 1},
    {"alpha": 1e-2, "T": 4, "N": 500, "pi": 0.5, "approx_factor": 1},
]

tqdm_test_inputs = tqdm(test_inputs, leave=True)

for input in tqdm_test_inputs:
    print("Optimal trade schedule for: ", input)
    print( approx_trade_schedule(**input) )
```

```
  0%|          | 0/5 [00:00<?, ?it/s]
Optimal trade schedule for:  {'alpha': 0.01, 'T': 4, 'N': 1000, 'pi': 0.
3, 'approx_factor': 1}
[120, 200, 340, 340]
Optimal trade schedule for:  {'alpha': 0.01, 'T': 3, 'N': 1000, 'pi': 0.
3, 'approx_factor': 1}
[190, 340, 470]
Optimal trade schedule for:  {'alpha': 0.1, 'T': 3, 'N': 2000, 'pi': 0.1,
'approx_factor': 1}
[220, 870, 910]
Optimal trade schedule for:  {'alpha': 1e-05, 'T': 4, 'N': 500, 'pi': 0.
5, 'approx_factor': 1}
[0, 0, 0, 500]
Optimal trade schedule for:  {'alpha': 0.01, 'T': 4, 'N': 500, 'pi': 0.5,
'approx_factor': 1}
[40, 160, 90, 210]
```

Task 2

Test the effectiveness of this computed schedule using the first 2 hours of each day in the TSLA data

To do so, we divide the first 2 hours of each day into 12 separate intervals of ten minutes each. Each interval is evaluated as follows. Suppose that the traded volume in that interval is given by the numbers $(V_0, V_1, \ldots, V_9)$. Then the interval score we assign to our schedule is given by

$$\sum_{i=0}^{9} \min\{S_i, V_i/100\}.$$

Effectively, this scheme allows us to trade up to a volume of 1% of what the market actually traded.

In [6]:
```
TRADE = pd.read_csv("./TSLA_TRADE.csv")

#Convert to datetime
TRADE['Dates'] = pd.to_datetime(TRADE['Dates'])
TRADE.set_index('Dates', inplace=True)

#Select only half the data and only volume
TRADE = TRADE["Volume"][:len(TRADE)//2]

# Select only first two hours
mask = (TRADE.index.hour >= 9) & ((TRADE.index.hour < 11) | ((TRADE.index
TRADE = TRADE[mask]

## Divide volume by 100 for Total score
# TRADE = TRADE/100

TRADE.to_csv("./TSLA_first_two_hours.csv")
```

In [7]:
```
df = pd.read_csv('./TSLA_first_two_hours.csv')

df['Dates'] = pd.to_datetime(df['Dates'])

df = df.set_index('Dates').sort_index()
```

In [8]:
```
x = df.index.min()
ub = df.index.max()
day = pd.DateOffset(days=1)

res = list()

alpha = 1e-3
pi = 0.5

with tqdm(total=(ub-x).days) as pbar:
    while x < ub:
        mask = (df.index >= x) & (df.index < x + day)
        slice = df[mask].iloc[1:]

        if len(slice) == 120:

            for i in range(0, 12):

                interval = slice[ i*10 : (i+1)*10 ]
                total_interval_vol = interval['Volume'].sum()

                schedule = approx_trade_schedule(alpha, 10, total_interva

                S = []
                m = 0
                sum = 0
                for t in range(10):
                    m = 0.1*m + 0.9*schedule[t]
                    S.append( (1 - alpha * (m**pi)) * schedule[t] )

                    interval_score = min( S[t] , interval.iloc[t]['Volume
                    sum += interval_score
                    # print(sum)

                res.append(sum)

        mask = slice = None
        x += day
        pbar.update(1)
```

```
  0%|          | 0/95 [00:00<?, ?it/s]
```

The TOTAL SCORE we assign to our schedule is the average of the all interval scores, averaged over the first 12 intervals of all the days in the first half of our data

In [9]:
```
print(f"Total score : {round(np.mean(res), 2)}")
```

```
Total score : 7676.96
```

Task 3

Code an algorithm that (approximately) does the following

1. It approximately enumerates all possible values for π between 0.3 and 0.7

2. It approximately computes the value of π that maximizes the TOTAL SCORE, when $N = 100000$, $T = 10$ and $\alpha = 0.001$.

3. This means that we run the DP algorithm (under the chosen value of π) and then evaluate as above to compute the TOTAL SCORE.

In [13]:
```
pi_range = np.round(np.linspace(0.3, 0.7, 20), 3)
tqdm_pi_range = tqdm(pi_range)
alpha = 1e-3
T = 10
N = 100000
for pi in tqdm_pi_range:
    schedule = approx_trade_schedule(alpha, T, N, pi, 3)
    S = []
    for t in range(10):
        m = 0.1*m + 0.9*schedule[t]
        S.append((1 - alpha * (m**pi)) * schedule[t])

    print(f"Score for {pi} --> {round( np.mean(S) , 2)}")
```

```
  0%|          | 0/20 [00:00<?, ?it/s]
Score for 0.3 --> 9770.6
Score for 0.321 --> 9720.41
Score for 0.342 --> 9678.05
Score for 0.363 --> 9609.23
Score for 0.384 --> 9554.35
Score for 0.405 --> 9486.2
Score for 0.426 --> 9397.37
Score for 0.447 --> 9299.58
Score for 0.468 --> 9206.12
Score for 0.489 --> 9095.37
Score for 0.511 --> 8866.1
Score for 0.532 --> 8578.18
Score for 0.553 --> 8335.77
Score for 0.574 --> 8021.77
Score for 0.595 --> 7547.28
Score for 0.616 --> 7087.29
Score for 0.637 --> 6368.6
Score for 0.658 --> 5616.54
Score for 0.679 --> 4251.33
Score for 0.7 --> 3382.07
```

In [ ]:
```
## Plot score v/s alpha
```