Dynamic Programming: Market Making 3 Ways

Zachary Barillaro - 20144016 - IFT 6251

April 8, 2020

1 Introduction

A market maker facilitates transactions of an underlying asset by providing purchase and sale prices at which market participants may execute their orders. The act of market making is not limited to financial markets as it is seen in many businesses such as vehicle sales, sports betting and more.

Lets imagine a world with 3 players: A car dealership, Logan the buyer and Marie the seller. Logan wishes to buy a car and Marie wishes to sell a car. Both participants do not know each other and wish to carry out any transaction with a registered dealership for security and product guarantee. The dealership acts as the middle man that buys the car from Marie and sells the car to Logan. The car dealership buys the car at a lower price and sells it at a higher price to compensate for the risk they bear. When all is said and done, both parties have completed their desired transactions and the dealership has earned a *spread* for facilitating the transaction. The *spread* is defined as the difference between the purchase price from Marie and the sale price to Logan.

In this project, I will be looking at market making inside financial markets. The concept of the market maker does not change from the example above, however, the frequency of transactions increase dramatically. We can imagine that most market making activity conducted today is executed via computerized algorithms to match the frequency of transactions. On any given financial instrument, there could be thousands of market makers posting their buy (bid) prices and sell (ask) prices based on their short-term expectation of the instrument and their inventory.

Market makers should behave with respect to their mandate to provide liquidity. This means they should not be taking a directional view on the underlying instrument, they should be transacting frequently and should always have a buy (bid) order and a sell (ask) order placed in the market. In fact, these are the strict requirements of the market maker I create in this paper.

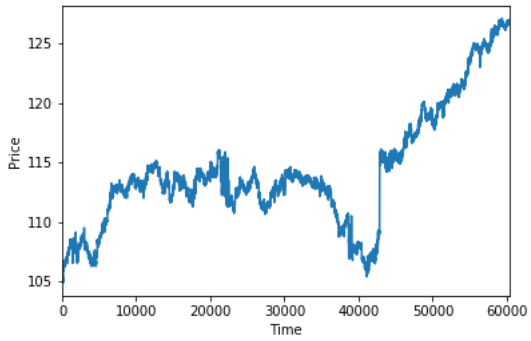

Figure 1: IBM Price Over Time

2 Modelling

2.1 Data

I used real transaction data taken from IBM stock.

	datetime	volume	bid	ask	price
0	90110134228	18800	105.375	105.50	105.375
1	90110134236	400	105.375	105.50	105.375
2	90110134236	100	105.375	105.50	105.375
3	90110134237	1000	105.250	105.62	105.375
4	90110134242	100	105.375	105.50	105.375

The data has 60,000 transactions over the date range 11/1/90-1/31/91, as suggested by the source. The price trend over this period is seen in Figure 1.

Financial instrument prices before the 2000s changed by increments of c dollars called *ticks*. For IBM, the tick value was 0.125$. Using this tick value and the change in price, a representation of the price series by changes in ticks is obtained. The transformed series is seen in Figure 2.

To smooth out the distribution and time series, I impose a maximum value on the tick change per period and redistribute the extreme values to the tails of the distribution. Ultimately, this helps to reduce the dimensionality of the state space. The transformed series is seen in Figure 3.

Last but not least, I compute the co-occurrence table with the IBM data to understand the following question: "Given the price change was -1, what is the probability distribution of the next price change?". Since it has been shown empirically[Analysis of Financial Time Series, 3rd Edition] that the lag-1 autocorrelation is significant in modelling transaction data, we include it in this

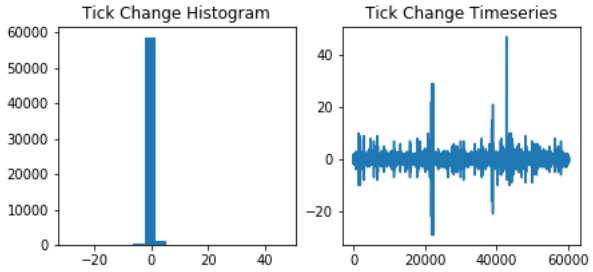

Figure 2: IBM Tick Change Series

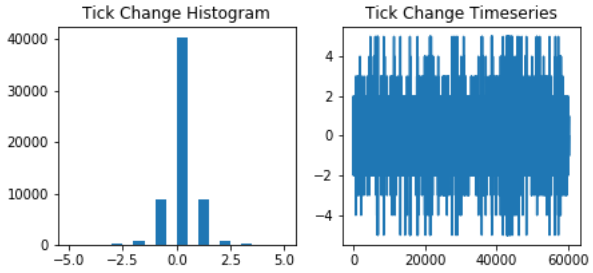

Figure 3: Smoothed IBM Tick Change Series

model.

	-4	-3	-2	-1	0	1	2	3	4
-4	0.00	0.00	4.21	7.37	35.79	14.74	7.37	12.63	17.89
-3	0.00	0.55	2.76	6.63	35.91	16.02	16.57	17.13	4.42
-2	0.11	0.33	1.54	4.63	40.02	29.99	20.73	1.54	1.10
-1	0.02	0.10	0.37	3.28	57.91	35.39	2.45	0.41	0.07
0	0.08	0.16	0.70	11.17	74.12	12.34	1.15	0.20	0.08
1	0.08	0.07	2.38	40.30	52.54	3.88	0.55	0.10	0.10
2	1.35	4.25	30.91	36.10	23.86	2.18	1.04	0.21	0.10
3	6.56	23.50	29.51	10.93	22.95	4.37	1.09	0.00	1.09
4	33.33	16.09	8.05	2.30	36.78	3.45	0.00	0.00	0.00

This table reads "Given the current price change was -1, the next price change being +1 has a probability of 35.39% and only 3.28% of being -1". The randomness that drives the stochastic sequential decision problem is derived from this table.

2.2 Dynamic Programming Formulation

2.2.1 States

The states are vectors of length 3,

$$x_k = (x_{1,k}, x_{2,k}, x_{3,k})$$

where $x_{1,k}$ is the current inventory of stock held by the market maker. $x_{2,k}$ is the unrealized profit or loss on the inventory that is held by the market maker. $x_{3,k}$ is the previous observed price change. To reduce the dimensionality of the state space, I restrict the the range of possible values to:

$x_{1,k} \in$ (-3,3)

$x_{2,k} \in$ (-30,30)

$x_{3,k} \in$ (-4,4)

2.2.2 Actions

The actions are vectors of length 4,

$$u_k = (u_{1,k}, u_{2,k}, u_{3,k}, u_{4,k})$$

where $u_{1,k}$ is the bid price, $u_{2,k}$ is the bid volume, $u_{3,k}$ is the ask price, $u_{4,k}$ is the ask volume. The range of values for each element are restricted to:

$u_{1,k} \in$ (-5,0)

$u_{2,k} \in$ (1)

$u_{3,k} \in$ (0,5)

$u_{4,k} \in$ (1)

I remove the action with the bid price and ask price set to zero. This is to avoid transacting with oneself. I did not experiment with more than 1 volume per transaction, however, the solution can easily be extended to include higher volumes per transaction. The careful reader will notice that an order price of +/- 5 is permitted when the maximum change in price is known to be +/- 4. This added tick is forced on the market maker to ensure no more contracts are held when the upper bound on inventory is met.

2.2.3 Revenue Function

The terminal revenue function, $g_N(\cdot)$, simply realizes the unrealized profit or loss which is the second vector element in the state.

$$g_N(x_N) = x_{2,N}$$

The revenue function, $g_k(\cdot)$, incurs revenues according to the mandate of providing liquidity, turning a profit and not holding risk. The market maker is compensated for closing a previous position (liquidity), however, a portion of the unrealized profit or loss is realized (profit). The market maker is penalized by a portion of the negative unrealized loss per period held (risk). The revenue depends on a random tick change in price denoted w_k.

$$g_k(x_k, u_k, w_k) = \begin{cases} \frac{x_{2,k}}{|x_{1,k}|} + 0.05 & \text{if } w_k \leq u_{1,k} \text{ and } x_{1,k} < 0 \\ \frac{x_{2,k}}{|x_{1,k}|} + 0.05 & \text{if } w_k \geq u_{3,k} \text{ and } x_{1,k} > 0 \\ -0.1x_{1,k}^2 & \text{if } x_{2,k} \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

The condition for a transaction to occur is that the random jump is equal to or exceeds the value of the bid or the ask prices. This concept comes from limit orders where a "price or better" scheme is employed.

2.2.4 Transition Function

The transition function, $x_{k+1} = f_k(x_k, u_k, w_k)$, is more complicated.
The function is broken up per vector element as:

- $$x_{1,k+1} = \begin{cases} x_{1,k} + 1 & \text{if } w_k \leq u_{1,k} \\ x_{1,k} - 1 & \text{if } w_k \geq u_{3,k} \\ x_{1,k} & \text{otherwise} \end{cases}$$
- $$x_{2,k+1} = \begin{cases} \lfloor (1 - \frac{1}{|x_{1,k}|})(x_{2,k} + x_{1,k}w_k) \rfloor & \text{if } w_k \leq u_{1,k} \text{ and } x_{1,k} < 0 \\ \lfloor (1 - \frac{1}{|x_{1,k}|})(x_{2,k} + x_{1,k}w_k) \rfloor & \text{if } w_k \geq u_{3,k} \text{ and } x_{1,k} > 0 \\ x_{2,k} + x_{1,k}w_k & \text{otherwise} \end{cases}$$
- $x_{3,k+1} = w_k$

2.2.5 Recurrence Equation

We want to maximize since we are dealing with a revenue and not a cost function.

$$J_k(x_k) = \max_{u \in U_k} \mathbb{E}_w \{g_k(x_k, u, w) + J_{k+1}(f_k(x_k, u, w))\}$$

Recall that w_k is driven by the co-occurrence matrix from the "Data" section above.

3 Algorithms

3.1 Evaluating the Expectation

The evaluation of the expectation in the recurrence equation is agnostic to the choice of algorithm if we assume that J_{k+1} is replaced with its direct or approximate form accordingly. Thanks to previous modelling choices, the expectation is computed by the average

$$\sum_w p(w_i|x_{3,k})z$$

where we sum over all possible tick values $w \in (-4, 4)$ and multiply the quantity of interest z by the associated conditional probability $p(w_i|x_{3,k})$. The conditional probability is derived from the co-occurrence matrix by conditioning on the most recent price change $x_{3,k}$. The quantity of interest z can be interpreted as $g_k(x_k, u_k, w_i) + J_{k+1}(f_k(x_k, u_k, w_i))$ with J_{k+1} replaced by J^*_{k+1} or $\hat{J}_{k+1}$.

Algorithm 1: Expectation Evaluation

```
values ← new storage
x ← random state
for u in U(x) do
    //All possible actions
    revenue ← 0
    for w in ticks do
        //All possible ticks
        revenue ← revenue + p(w|x_3)(g_k(x, u, w) + J_{k+1}(f_k(x, u, w)))
    end
    values[u] ← revenue
end
max_revenue ← max(values)
max_action ← argmax(values) // Use the index on the action space to
 get the action.
```

I will use max_revenue and max_action throughout other pseudo-codes to denote the operation of evaluating the expectation and obtaining the max value and associated action. For each algorithm, I will specify J_{k+1} accordingly.

3.2 Direct Method

Algorithm 2: Direct Method Pseudo-Code

$g \leftarrow g_N$
$J_N^* \leftarrow$ new storage
for x_n *in* X_N **do**
  $J_N^*(x_n) = g_N(x_n)$
end
$k \leftarrow K$
$g \leftarrow g_k$
$J_{k+1}^* \leftarrow J_N^*$
$J_k^* \leftarrow$ new storage
$U_k^* \leftarrow$ new storage
while $k \geq 0$ **do**
  for x_k *in* X_k **do**
    $J_k^*[x_k] =$ max_revenue
    $U_k^*[x_k] =$ max_action
  end
  $J_{k+1}^* \leftarrow J_k^*$
  $J_k^* \leftarrow$ new storage
  Save U_k
  $U_k^* \leftarrow$ new storage
  $k \leftarrow k - 1$
  $g \leftarrow g_k$
end

I define J_{k+1} here as J_{k+1}^*, the optimal J found through back recursion starting at stage N.

3.3 Approximate Form: Rollout

For rollout, I create a recursive function that walks through many sample paths and calculates the expected value along each path. Given that we start this algorithm at every possible next state x_{k+1} given the current state x_k, we can

approximate the cost-to-go function J_{k+1}. This approximation is $\hat{J}_{k+1}$.

Algorithm 3: Monte Carlo Path Recursion - Function: deeper(x, totalcost, k)

```
if k == 2 then
    return totalcost
else
    U = subset(U_k(x))
    W = sample(p(·|x_3))
    for u in U do
        for w in W do
            x_{k+1} = f_k(x, u, w)
            cost_{+1} = g_k(x, u, w)
            deeper(x_{k+1}, totalcost + p(tick|x)cost_{+1},k+1)
        end
    end
end
```

The recursive function recurses 2 steps ahead by sub-sampling the action space to reduce the complexity. It also randomly samples tick values to focus the computational resources on probable scenarios. Each of the costs are weighted by the associated probability, summed and returned once the particular path is completed. Since rollout is online, the policy is computed over a particular path as follows:

Algorithm 4: Rollout Algorithm: $\hat{J}_{k+1}$ via Monte-Carlo Averaging

```
path ← random_path //Array of tick values
x = (0,0,0)
for tick in path do
    best_action =
     arg max_{u∈U_k} E_w {g(x̂_k, u, w) + deeper(f_k(x_k, u, w), 0, 0)}//Deeper is
     defined as above to estimate the cost-to-go function
    x = f_k(x, best_action, tick)//Go to next state
end
```

3.4 Approximate Form: Parametric Approximation

For parametric approximation, I first generate 10,000 sample paths that are created by using the co-occurrence matrix and random sampling. I then run a random market maker through each path where the action picked at each stage is chosen randomly. This yields a set of states that should mirror the natural

distribution of states visited per stage. I'll call the random set of states $\hat{X}_k$.

Algorithm 5: Parametric Approximation - Linear Regression

$g \leftarrow g_N$
$X \leftarrow$ new storage
$y \leftarrow$ new storage
for $\hat{x}_n$ *in* $\hat{X}_N$ **do**
  $X[\hat{x}_n] = \hat{x}_n$
  $y[\hat{x}_n] = g(\hat{x}_n)$
end
$r_N = LinearRegression.solve(X, y)$ //Linear Regression Parameters for stage N
$k \leftarrow N - 1$
$g \leftarrow g_k$
$r \leftarrow r_N$
while $k \geq 0$ **do**
  $X \leftarrow$ new storage
  $y \leftarrow$ new storage
  for $\hat{x}_k$ *in* $\hat{X}_k$ **do**
    $\mathrm{X}[\hat{x}_k] = \hat{x}_k$
    $\mathrm{y}[\hat{x}_k] = \max_{u \in U_k} \mathbb{E}_w \{g(\hat{x}_k, u, w) + \hat{J}_{k+1}(f_k(\hat{x}_k, u, w), r)\}$
  end
  $r_k \leftarrow LinearRegression.solve(X, y)$ //Linear Regression Parameters for stage N
  $r \leftarrow r_k$
  $k \leftarrow k - 1$
  $g \leftarrow g_k$
end

The algorithm generates a sequence of N parameters vectors that are used by the linear regression model to estimate the cost-to-go function. This framework can be extended to neural networks or other models easily.

4 Results & Analysis

Refer to the policies/ directory in the project folder for a visualization of policies generated by the algorithms.

4.1 Instance Details

The instances I use to evaluate the performance of each algorithm above differ by the number of prices changes. The small instance has 50 price changes, the medium instance has 100 price changes and the large has 200 price changes. Although the number of price changes describes the possible range of prices, it does not guarantee that different price dynamics are used for evaluation. For this reason, I pick 6 paths per instance that cover the volatility, trending and

calmness dynamics. See Figure 4. Each model will be benchmarked against these dynamics to understand which ones cause the market maker to fail and which ones cause the market maker to prosper.

4.2 Complexity

	back_recursion	approximation	rollout
50	181.37	37.02	196.35
100	376.23	79.30	615.62
200	704.34	170.33	1168.09

The parametric approximation models are the least computationally complex when only considering the fitting of each stage's model. A crucial part of this process is the Monte-Carlo sampling; the Monte-Carlo step takes around 200 seconds. Rollout is misrepresented here as the time displayed is only for a single path. One can multiply the time by 6 to understand the computational requirements of running an instance over all dynamics when using rollout.

4.3 Small Instance Analysis

Recall that the market makers mandate is to provide liquidity by transacting frequently while turning a profit and holding small amounts of risk. Rather than explicitly reporting profit, I will use the total reward which accounts for profit, risk penalties and trade compensations. This means the market maker is penalized for holding risk at any point, is compensated for closing a previously opened trade and is rewarded/penalized for any profit/loss generated.

Path Name	uptrend	downtrend	no_change	high_vol	low_vol	avg_vol
Num. Trades	50.00	42.00	48.00	48.00	48.00	50.00
Closing Trades	50.00	40.00	48.00	47.00	48.00	50.00
Continuation Trades	0.00	2.00	0.00	1.00	0.00	0.00
Max Runup	0.00	0.00	0.00	0.00	0.00	0.00
Max Drawdown	0.00	-13.00	-1.00	-1.00	-1.00	0.00
Most Held Position	0.00	1.00	1.00	1.00	0.00	0.00
Most Held Position by %	50.98	52.94	49.02	47.06	50.98	50.98
Least Held Position	-1.00	2.00	-1.00	2.00	1.00	-1.00
Least Held Position by %	7.84	3.92	1.96	1.96	49.02	3.92
Total Revenue	16.25	-17.18	8.00	27.10	0.10	10.25
Avg Revenue per Period	0.32	-0.34	0.16	0.53	0.00	0.20

I wont break down every metric and I will only present highlights going forward but these are the metrics I will use for my analysis. It gives a birds eye view of the policy and how the policy changes given the underlying dynamics of the path. The best metric is the "Avg Revenue per Period" as it is comparable to any instance, algorithm and path combination.

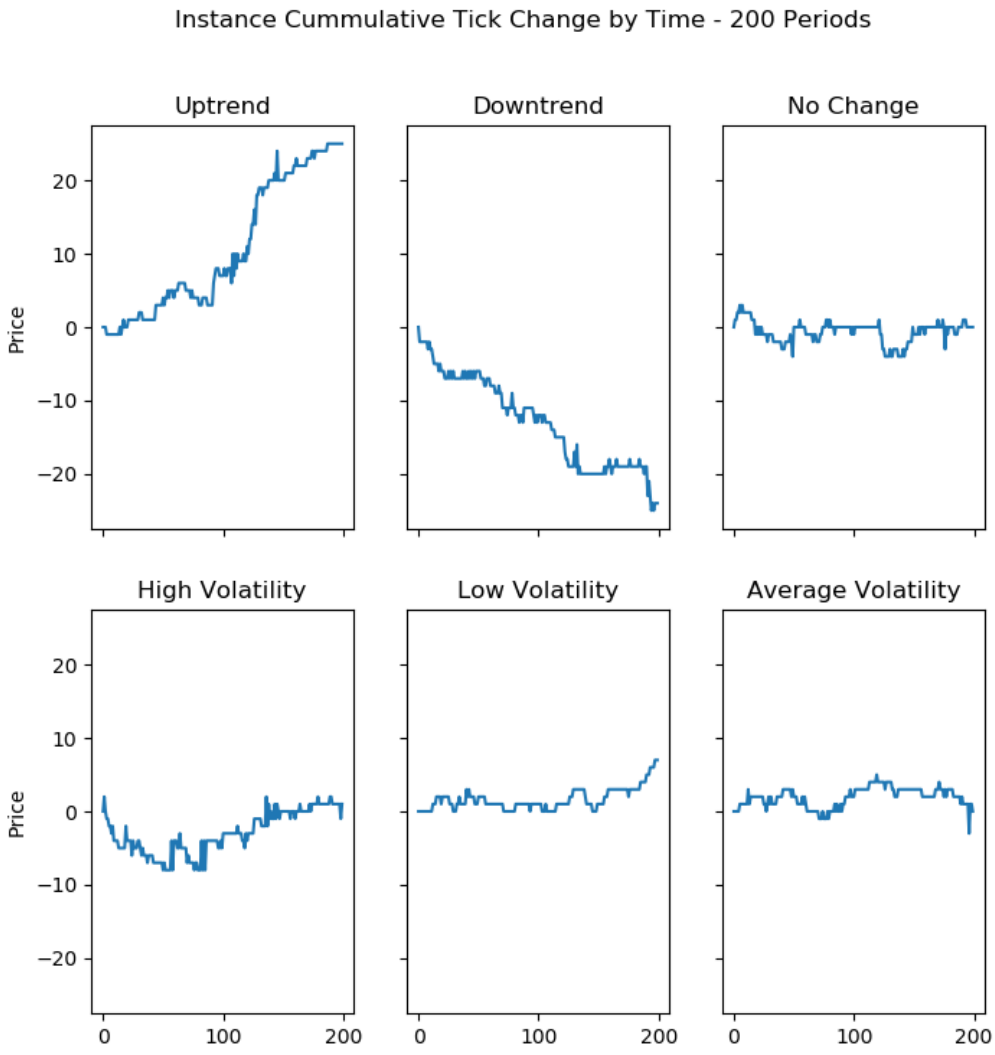
Instance Cummulative Tick Change by Time - 200 Periods
Uptrend
Downtrend
No Change
High Volatility
Low Volatility
Average Volatility
Price
Price
20
10
0
−10
−20
0
100
200

Figure 4:

Looking back at Figure 1, there is an obvious positive trend in the data that can be learned by the algorithms. It seems like back recursion has done just this and is seen in the "Most Held Position" statistic where the most held position is 1 or 0. This means the algorithm likes to hold 1 long stock, sells it and buys it back.

"Continuation Trades" indicates the number of times the algorithm adds to an existing position rather than closing the position. Since the values are so low, it means the algorithm does not like to hold much risk. This is the intended behavior. There is enough evidence to suggest that the modelling choices are correct and working. Since the high_vol path has the most profit, I'll focus on that path first when comparing algorithms.

4.4 Medium & Large Analysis

4.4.1 High Vol Path - Horizon 100

	Back Recursion	Rollout	Approximation
Num. Trades	94.00	68.0	60.00
Most Held Position	0.00	1.0	1.00
Most Held Position by %	48.51	47.0	68.32
Least Held Position	-1.00	-1.0	0.00
Least Held Position by %	3.96	12.0	31.68
Total Revenue	28.85	30.3	23.60
Avg Revenue per Period	0.29	0.3	0.23

What interested me the most when comparing all the algorithms against the high vol path was the difference in number of trades taken. Back Recursion nearly had 1 trade per period where Rollout and Approximation seemed to be less active. Despite that, Rollout had better "Total Revenue" and was more diverse in its position selection. It held 1 short position 12% of the time versus 3.96% for Back Recursion. Since Rollout is locally optimized and only uses a 2-step look ahead, it becomes less biased by the long term uptrend seen in the data. Approximation has the highest positive bias with a long position held 68.32% of the time which clearly does poorly in a highly volatile environment. Rollout is a clear winner here.

4.4.2 Avg Vol Path - Horizon 200

	Back Recursion	Rollout	Approximation
Num. Trades	191.00	152.00	154.00
Closing Trades	191.00	151.00	153.00
Most Held Position	0.00	0.00	1.00
Most Held Position by %	48.76	40.50	57.71
Total Revenue	27.05	18.40	20.55
Avg Revenue per Period	0.13	0.09	0.10

The "avg_vol" path should be the most "normal" path of all those selected. When this is the case, all algorithms within a few cents of each other when considering the average revenue per period. Unlike last period, the more frequent trading paid off for the Back Recursion algorithm. It received only 2$ of extra trade compensation for the 40 extra trades meaning it also made better decisions along the way.

4.4.3 Uptrend Path - Horizon 100

	Back Recursion	Rollout	Approximation
Num. Trades	100.00	7.00	78.00
Most Held Position	0.00	-1.00	1.00
Most Held Position by %	50.50	46.00	53.47
Total Revenue	22.50	-32.90	17.45
Avg Revenue per Period	0.22	-0.33	0.17

This reiterates approximation's long bias as holding a long position in an uptrending market is favorable. The 2-step look ahead approach taken by Rollout realized a net loss, however, is the expected behavior. Since these trends have such low probability, Rollout chooses to bet against any jump in price as the co-occurrence matrix suggests to do. Rollout took the probabilistic approach but realized a net loss in the process. This is the point of choosing these outlier scenarios.

4.4.4 Downtrend Path - Horizon 200

	Back Recursion	Rollout	Approximation
Num. Trades	186.00	5.00	4.00
Most Held Position	1.00	3.00	1.00
Most Held Position by %	46.77	42.50	66.17
Total Revenue	17.02	-112.45	-52.55
Avg Revenue per Period	0.08	-0.56	-0.26

By the same token, Rollout's probabilistic betting and Approximation's long bias are two strategies that do not perform well in the downtrend path. It surprised me how much the two approximate algorithms broke down and barely traded in this scenario. I would have expected them to be a little more adaptive and trade their way out of the losses. It seems that Rollout bought so many shares that it hit the 3 share limit and did not come to the conclusion that selling the shares was the best thing to do. It traded off realizing a huge unrealized loss or receiving the risk penalty per period. Not the behavior I intended but still within the confines of how these algorithms behave. As for Approximation's strategy, it seems there's a recurring theme of just holding 1 long share and profiting from that. There could be a situation where the parameters are over-fit and have learned that this strategy, on average, is the best. The "Avg Revenue.." for Back Recursion is small but still positive as it has been for every other instance. Its clear that back recursion is the optimal solution despite

losing to rollout in the high volatility path. It has been consistent through all other dynamics and horizons that were tested.

5 Conclusion

Many modelling decisions have come together to create 3 different market makers that behave very differently given different price dynamics. 60,000 data points with a positive trend were used to create the randomness that drove the sequential decision problem. We later learn that this positive bias influenced the approximate methods' decision making. Further work can be done on refining the data that is used to smooth out the biases. Outside of the report, I did a lot of testing with different values for compensating trades and penalizing risk. It turns out that these values greatly affect how the algorithms behave and deserve close attention to their design. I settled on the values given in the report as they offered a wide range of results.

We learned that the back recursion algorithm did consistently well across all market dynamics as the "Average Revenue per Period" was always positive. It did so by adopting a very frequent trading strategy that minimized the holding risk and stuck to its mandate very closely, to provide liquidity.

I invite any and all readers to watch the policy visualizations in the policies/ directory of the project.

6 Bibliography

Inspiration of the idea: Chapter 5 of Analysis of Financial Time Series, 3rd Edition by Ruey S. Tsay, Wiley

Self-Assessment

3.1 Problem Description

A- to A+. I opted for a problem that hasn't been seen in class and that needed real world data to be solved. It was not easy but it was fun!

3.2 Modelling

A- to A+. I included notation that was consistent with the notation seen in class and in books. I explained my modelling choices and broke down each element of the dynamic programming solution.

3.3 Algorithms

A- to A+. I provided pseudo-code where needed and clearly explained how to expectation is evaluated in each problem.

3.4 Results and Analysis

A- to A+. I hand picked specific instances which showed where the algorithms failed and gave possible explanations as to why this happens. Good results across all algorithms when given the "average" scenario. I chose metrics that can be compared across algorithms and offered different metrics dependent on the case/instance being analyzed.

3.5 Conclusion

A- to A+. Short but highlights the findings and where further work can be focused on to increase the results of the solution.

3.6 Other parts

A- to A+. Complies with instructions.

3.7 Code

A- to A+. Complies with instructions.

3.7 Poster

A- to A+. I chose to give a purely illustrated view of the project. It is meant to be understood at a birds eye view. I'm not sure if this was the expected poster but it complements the paper well.