FYS3150-Project5
Stock Market model
Chenghao Liu
December 5, 2018

## 1. Introduction

The purpose of this project is to simulate the stock market model based on the Monte Carlo method, simulating transactions between different financial agents, with the ultimate goal of obtaining a distribution function based on the capital income m when balancing. From Pareto's work (V. Pareto, 1897) it is known from influenced studies that the higher end of the distribution of money follows a distribution:

$$\omega_m \propto m^{-1-a}$$

And a is between 1 and 2.

In this project, we assume that there are N agents, which will conduct many transactions in the middle. In order to achieve balance, we choose to simulate one million times. In this way, each agent can conduct sufficient transactions to achieve a balance. The rules of the transaction are as follows: the money owned by both agent before the transaction are mi and mj. By generating a random number x of 0~1, the two agent after the transaction are redistributed, that is, the money owned by the two agent after the transaction are respectively x(mi+mj) and (1-x)(mi+mj) , the amount of the total transaction is (1-x)mi-xmj. A relatively stable dynamic balance will be achieved among the N agents, and the distribution of money owned by each stock market agent will form a balance function, which is theoretically the Gibbs distribution:

$$\omega_m = \beta exp(-\beta m)$$

And beta is the reciprocal of the average money for N agents.

The project is divided into five parts. The first part is to simulate the above transaction process. After one million simulations, the distribution data of the money is obtained. The distribution function image is drawn at intervals of 0.01~0.05, and the Gibbs distribution is compared. The second part is to take wm as the natural logarithm to see if the image is a straight line. The third part is to add a savings factor z. Suppose each agent does not take out

all the money to trade, but leaves a part, trades with the remaining money, and obtains balanced data after multiple simulations. The fourth part is the nearest neighbor interaction, that is, considering the reality, each agent may prefer to trade with stock market agents of similar size, so add a probability pij:

$$p_{ij} \propto |m_i - m_j|^{-a}$$

The simulations were carried out when N was at 500 and 1000, respectively, and the distribution of money at a time of 0.5, 1.0, 1.5, and 2.0 was examined.

The fifth part is to combine the nearest neighbor interaction with the former transaction, that is, in addition to the above part, each stock market agent is more inclined to trade with the company that it has previously traded, which requires a python dictionary to be generated one million times. The information of the transaction is recorded, the number of transactions between the various agents is obtained, and the coefficient cij is obtained, so the pij becomes as follows:

$$p_{ij} \propto |m_i - m_j|^{-a}(c_{ij} + 1)^{\gamma}$$

Cij is the number of previous transactions for agency i and agency j. Simulations were performed at N=500 and 1000, a=1.0 and 2.0, γ=0.0, 1.0, 2.0, 3.0, 4.0, respectively, to obtain a money distribution image.

I used the Monte Carlo method to simulate this project. The algorithm of the code can refer to the Method Part below.

The program can be found in https://github.com/lch172061365/Computational-Physics.git

Due to the limitation of Python's own calculation accuracy and the fluctuation of the dynamic balance of the algorithm itself, the image is more scattered when the step is smaller. Only the general trend can be seen. Adjusting the step to a larger value can make the image clearer and more intuitive. You can observe the trend of its changes. In general, the program is reliable and convincible, and the results are relatively correct. We obtained the various images needed for 5a to 5e, and obtained the money distribution in the final dynamic equilibrium in the stock market model by changing various parameters such as z, y, N.

In 5a we obtained a money distribution in the form of an exponential decline. In 5b we can observe that log(wm) decreases linearly with m. After adding the addition of the index

index z in 5c, a peak appears at z*m. After adding the nearest neighbor index in 5d, the drop would indeed be like the form of m^-1-a. In 5e, it was found that after adding the former transaction item, several small peaks appeared in the image, and some changes occurred.

## 2. Method

This project uses the Monte Carlo Method for simulation and uses Python as the programming software. Python's various libraries can meet the needs of the project and solve its problems. The procedures for the entire project are roughly as follows, and the programs under different problems will be adjusted as needed:

```
import random
import matplotlib.pyplot as plt
import math

m0 = 1 #initial money
N = 500
agent_moneylist = [m0]*N #500 agents intial money list

#monte carlo algorithm, random agent and radom transaction index x
def trade(agent_moneylist):
    x = random.uniform(0,1) #random trading money percent
    trade_agent = random.sample(range(0,N),2) #random two agents
    ta1 = trade_agent[0]
    ta2 = trade_agent[1]
    money_ta1 = x*(agent_moneylist[ta1] + agent_moneylist[ta2])
    money_ta2 = (1-x)*(agent_moneylist[ta1] + agent_moneylist[ta2])
    agent_moneylist[ta1] = money_ta1
    agent_moneylist[ta2] = money_ta2
    #trade finish

def financial_engineering(agent_moneylist):
    for i in range(0,1000000):
        trade(agent_moneylist)
    return agent_moneylist

def data(agent_moneylist):
    agent_moneylist.sort()
    step = 0.05
    i = 1
    k = 1
    count = []
    money = []
    judgement = math.floor(agent_moneylist[0]*100)/100
    while i < N :
        if agent_moneylist[i] < judgement + step:
            k += 1
            i += 1
        else:
            if k != 0:#ignore those k=0 to make the figure looks better
                count.append(k)
                money.append(judgement)
                judgement += step
                k = 0
            else:
                judgement += step
                k = 0
    return count,money
```

```
def plot_making():
    count,money = data(agent_moneylist)
    x_values = money
    y_values = count
    plt.figure('Results for money trade')
    plt.title('money distribution for financial engineering')
    plt.xlabel('money')
    plt.ylabel('number')
    plt.scatter(x_values, y_values, s=20, c="#ff1212", marker='o')
    plt.show()


if __name__ == '__main__':
    financial_engineering(agent_moneylist)
    plot_making()
```

Figure 1 Main code for project5

The whole program includes four functions.

The first one is the most important trading function, that is, the Monte Carlo method is used to simulate a transaction. First, a random transaction factor x of 0~1 is generated, and then two number from 0 to N are randomly generated. The integers i and j, representing the two trading agents, conduct a transaction in accordance with the method described in Introduction, and update the post-trade money to the list.

The second function is a repeat function that takes the entire transaction a million times.

The third function is the data processing function. After the simulation is finished, the list of money obtained is classified according to the step, and the number of agents in a step is counted, which is used to represent the probability distribution of money, and the list of output money and counts. Used for drawing.

The fourth function is a drawing function. The resulting two lists are drawn by scatter plots using matplotlib to obtain a distribution image. There are different requirements for different parts of the project, and the above code can be adjusted, added or modified to achieve the purpose of the problem. There are four parameters in the program, m0 is the initial capital of the stock market agent, N is the total number of agents, step is the classification of money during data processing and analyzing, and the number of simulations is one million.

## 3. Implementation

### 5a

This part of the project simulates the transaction, simulating a million transactions from

the initial money m0 and getting a distribution of the final money. The code of this task is the code shown in the previous Introduction section. After running the code, the program first runs the financial_engineering function, which generates one million simulations and returns a list of simulation results. Then run the drawing function, first perform data processing, get two lists of counting and money, and then use matplotlib for simulation drawing. The pdf of the project description requires that the step should be taken between 0.01~0.05, but in the actual operation, it is found that the scatter plot is very scattered when it is taken at 0.05, and only the form of exponential decay can be roughly seen. When it is large, for example, when it is 0.2 or 0.4, the form of its exponential decline is very obvious. The figure below is an image with steps of 0.05, 0.2, and 0.4, respectively.

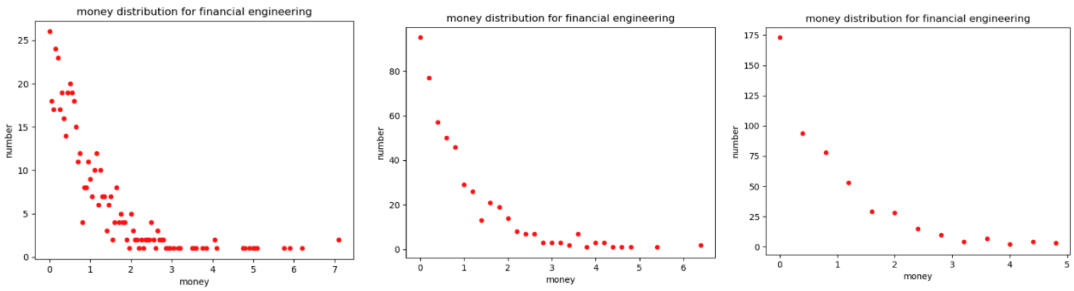



Figure 2 Money distribution(step at 0.05, 0.2 and 0.4, from left to right)

As shown in the above figure, it can be seen that the probability of capital distribution decays with the growth index of money.

## 5b

The purpose of the second project is to take the probability distribution of the money obtained in the previous part as the natural logarithm and then draw. Since the upper part is the number of the count, the probability distribution should be obtained by dividing N, and the logarithm is required. So we need minus it with log(N) and make a slight modification to the above code:

```
#wm = number/N, as known as number/500, so log(wm)=log(number)-log(N)
def recognizing_distribution():
    count,money = data(agent_moneylist)
    print(count)
    logwm_list=[0]*len(count)
    for i in range(len(count)):
        logwm_list[i] = math.log(count[i])-math.log(N)
    print(logwm_list)
    x_values = money
    y_values = logwm_list
    plt.figure('Recognizing Distribution')
    plt.title('money distribution for financial engineering')
    plt.xlabel('money')
    plt.ylabel('log(wm)')
    plt.scatter(x_values, y_values, s=20, c="#ff1212", marker='o')
    plt.show()
```

Figure 3 code for ecognzing distribution

After modification, the corresponding logwm_list is obtained, and the y_value of the drawing is used to draw the scatter plot. As in the case of 5a, when the step is taken at 0.05, the drawn image will be scattered due to the deviation caused by the dynamic fluctuation. So larger step can get a better, more intuitive image. The following figure is the log (wm) distribution image of step at 0.1, 0.2 and 0.4, respectively. When the step is large, the tail is a more obvious straight line.

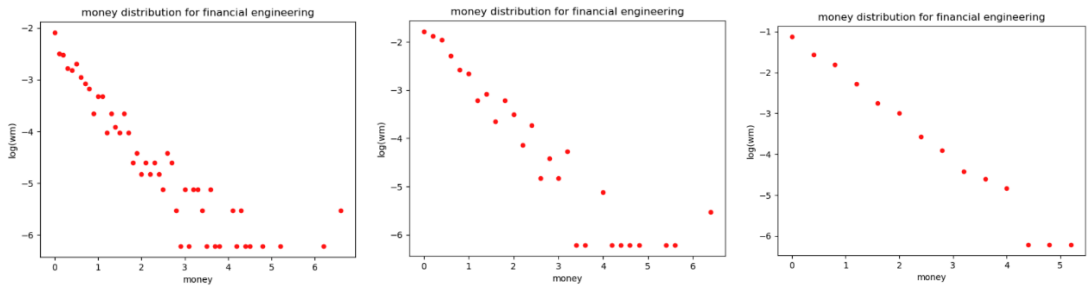



Figure 4 log(wm) distribution(step at 0.1, 0.2 and 0.4, from left to right)

## 5c

The purpose of the third task is to examine the transaction with saving index. It is assumed that each agent does not participate in all of its own money, but leaves a part of it to save. This requires a new addition of a saving index in the program. After adding the saving index z, the money involved in the transaction changed from (mi+mj) to (1-z)(mi-mj), so the money of the two agents i and j after the transaction become:

$$m_i^{`} = zm_i + x(1-z)(m_i + m_j)$$

$$m_j^{`} = zm_j + (1-x)(1-z)(m_i + m_j)$$

The simulation was performed when z was at 0.25, 0.5, 0.9, and an image was obtained. For the content of the above task, you can make the above modifications to the code.

```
m0 = 1 #initial money
N = 500
agent_moneylist = [1]*N #500 agents intial money list
z = 0.5 #saving index

#monte carlo algorithm, random agent and radom transaction index x
def trade(agent_moneylist):
    x = random.uniform(0,1) #random trading money percent
    trade_agent = random.sample(range(0,500),2) #random two agents
    ta1 = trade_agent[0]
    ta2 = trade_agent[1]
    money_ta1 =z*agent_moneylist[ta1]+x*(1-z)*(agent_moneylist[ta1] + agent_moneylist[ta2])
    money_ta2 =z*agent_moneylist[ta2]+(1-x)*(1-z)*(agent_moneylist[ta1] + agent_moneylist[ta2])
    agent_moneylist[ta1] = money_ta1
    agent_moneylist[ta2] = money_ta2
    #trade finish
```

Figure 5 adjusted code of 5c

The saving index is z, which can be modified in the program, and the trade function also modifies the trading formula according to the requirements of the task. Similarly, I found that when the step is at 0.05, the step is too small. Due to dynamic fluctuations, this step will cause the scatter figure to be very scattered. Therefore, the image of larger step is more intuitive and obvious. The experiment finds that the step is better at 0.3, when z=0.5; better at 0.2, when z=0.25; better at 0.1, when z=0.9. The following is the comparison of two steps: 0.05 and 0.2 when z = 0.5:

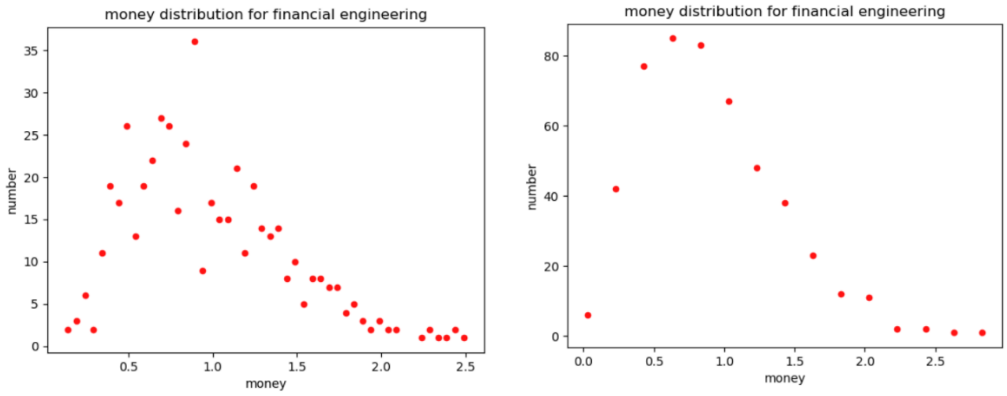



Figure 6 comparison figure of two steps: 0.05 and 0.2 when z = 0.5

Adjust z to 0.25, 0.5, 0.9 to get the different images required by the task.

## 5d

The fourth task is the nearest neighbor transaction, and it is believed that each agent will be more inclined to make a deal with those stock market agents which are close to its own size. Here, pij is introduced as a new parameter.,

$$p_{ij} \propto |m_i - m_j|^{-a}$$

This represents the probability of successful trading between the two agents. When mi and mj are similar, this probability will be very large, and possibility of the transaction is equivalent to 100%. So we can make the following adjustments and additions to the code:

```
dif = 0.004
a = 0.5
k = dif**a

def pij(mi,mj):
    pij = k/((abs(mi-mj)**a)+0.000001)
    #in case mi-mj=0, and 0.000001 is small enough to ignore the effect
    p = random.uniform(0,1)
    if p <= pij:
        return True  #Trasaction success
    else:
        return False #Transaction fail
```

Figure 7 adjusted code for 5d

Let me explain this code. The function pij of this code returns a bool value, indicating whether the transaction is successful. Dif is the money difference factor between the two agents, a is the coefficient factor required by the task. After I have passed the simulation of one million times, the list of money obtained indicates that the difference between the money of the two agents is the closest to 0.004, and The formula of pij indicates that the left term is proportional to the right term, and the right term reaches the maximum when (mi-mj) is 0.004. At this time, the pij should be 100%, so I set the dif to 0.004, so that the pij is proportional to the right term with a coefficient k.

At the same time, I considered that the money of each agent were the same at the beginning, so the difference is 0, which will lead to the calculation of errors, so I will add a minimum of 0.000001, which is small enough to not affect to the denominator, nor does it make the denominator zero, causing calculation errors.

This function generates a random number method to indicate the probability of success or failure of the transaction. If the generated random number is less than pij, it means the transaction is successful. If it is greater than pij, it means the transaction fails. Therefore, if pij is greater than 1, it can be considered that the transaction is 100% successful. Add the function pij to the trade function, you can complete the simulation of projec5d.

```
#monte carlo algorithm, random agent and radom transaction index x
def trade(agent_moneylist):
    x = random.uniform(0,1) #random trading money percent
    trade_agent = random.sample(range(0,N),2) #random two agents
    ta1 = trade_agent[0]
    ta2 = trade_agent[1]
    mi = agent_moneylist[ta1]
    mj = agent_moneylist[ta2]
    if pij(mi,mj) == True:
        money_ta1 =x*(agent_moneylist[ta1] + agent_moneylist[ta2])
        money_ta2 =(1-x)*(agent_moneylist[ta1] + agent_moneylist[ta2])
        agent_moneylist[ta1] = money_ta1
        agent_moneylist[ta2] = money_ta2
    else:
        pass
    #trade finish
```

Figure 8 adjusted code for trade function of 5d

By modifying a, we can obtain an image of the distribution of money when a is 0.5, 1.0, 1.5, and 2.0, respectively. Similar to 5c, we can add savings index z to obtain a distribution of nearest neighbor transaction money with and without savings. The result image can be found in Results Part.

### 5e

This part is Nearest neighbors and former transactions. As explained in the previous section, each agent prefers to trade with an agent of similar size, and tends to trade with a previously traded agent, that is, an item is added to the pij. The newly added pij is as follows:

$$p_{ij} \propto |m_i - m_j|^{-a}(c_{ij}+1)^{\gamma}$$

Cij is the number of transactions between i and j agents, which is equivalent to a transaction record, so this requires us to store one million transactions as a python dictionary to see how many transactions have been made before each agent. So the code in 5d needs to be modified and adjusted.

```
m0 = 1 #initial money
N = 1000
agent_moneylist = [1]*N #500 agents intial money list
dif = 0.0004
a = 1
k = dif**a
y = 1  #former transactions index
Trading_history = {}

def pij(mi,mj,key):
    pij = k*(Trading_history[key]**y)/(((abs(mi-mj)**a)+0.000001))
    #in case mi = mj,and 0.000001 is small enough to ignore the effect
    p = random.uniform(0,1)
    if p <= pij:
        return True  #Trasaction success
    else:
        return False #Transaction fail
```

Figure 9 adjusted code for pij function

As shown above, at this point we have a former transaction index y, so that we can adjust y in the code to take different values, such as 0.0, 1.0, 2.0, 3.0. Secondly, I created a python dictionary, trading_history, so the records of the transaction will be stored in this python dictionary. The key is two trading agents, arranged by the size of the serial number. The value is the number of transactions. This python dictionary is used to find the number of transactions between various agents.

I made some changes in the code of pij and added the Trading_history[key]**y item. There is no need to add 1 here because there is an added 1 in the Trading_history itself. The code for Trading_history is as follows

```
#monte carlo algorithm, random agent and radom transaction index x
def trade(agent_moneylist):
    x = random.uniform(0,1) #random trading money percent
    trade_agent = random.sample(range(0,N),2) #random two agents
    ta1 = trade_agent[0]
    ta2 = trade_agent[1]
    key = tuple(sorted(trade_agent))
    #if trade_agent in Trading_history:value+=1, else add this item, value=1
    if key in Trading_history.keys() == True:
        Trading_history[key] += 1
    else:
        Trading_history.update({key:1})
    mi = agent_moneylist[ta1]
    mj = agent_moneylist[ta2]
    if pij(mi,mj,key) == True:
        money_ta1 =x*(agent_moneylist[ta1] + agent_moneylist[ta2])
        money_ta2 =(1-x)*(agent_moneylist[ta1] + agent_moneylist[ta2])
        agent_moneylist[ta1] = money_ta1
        agent_moneylist[ta2] = money_ta2
    else:
        pass
    #trade finish
```

Figure 10 the code for trading_history in trade function

For each transaction's stock market agents i and j, they are sorted by size and become

key: (i,j), and then they are searched by key in the python dictionary, and the value obtained is the number of transactions. If they are in the python dictionary, then we can get the value: Trading_history[key]+1. If they are not in the python dictionary, we need to update the two trading agents and set the new value to 1.

With the above code, we can start to simulate the content of 5e. By modifying each parameter factor in the program, we can get distribution image when N=1000, a=1.0 and 2.0, y=0.0, 1.0, 2.0, 3.0 and 4.0. At the same time, the equilibrium index z can be added to obtain images of the distribution of money in the case of the saved index and the absence of the saved index. Typically, the distribution of money at N=1000, a=2, y=1, z=0.5 is as follows:

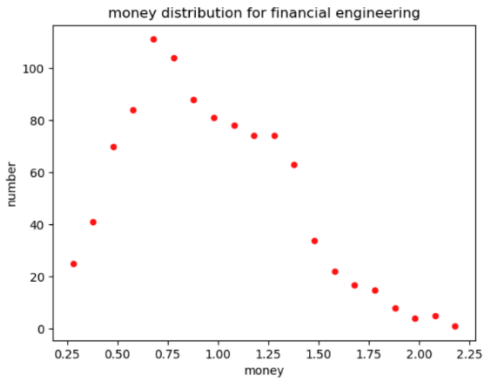



Figure 11 the distribution of money at N=1000, a=2, y=1, z=0.5

## 4. Results

### 5a

We obtained the image of the distribution of money of each agent after simulating a million times. The step required by the project is in the range of 0.01-0.05. After the experiment, it is found that the form of the exponential decline of the step at 0.2 or 0.4 is more clear and more neat. The following figure is the image of the money distribution when the step is at 0.05, 0.1, 0.2, and 0.4 (from left to right, up to down)

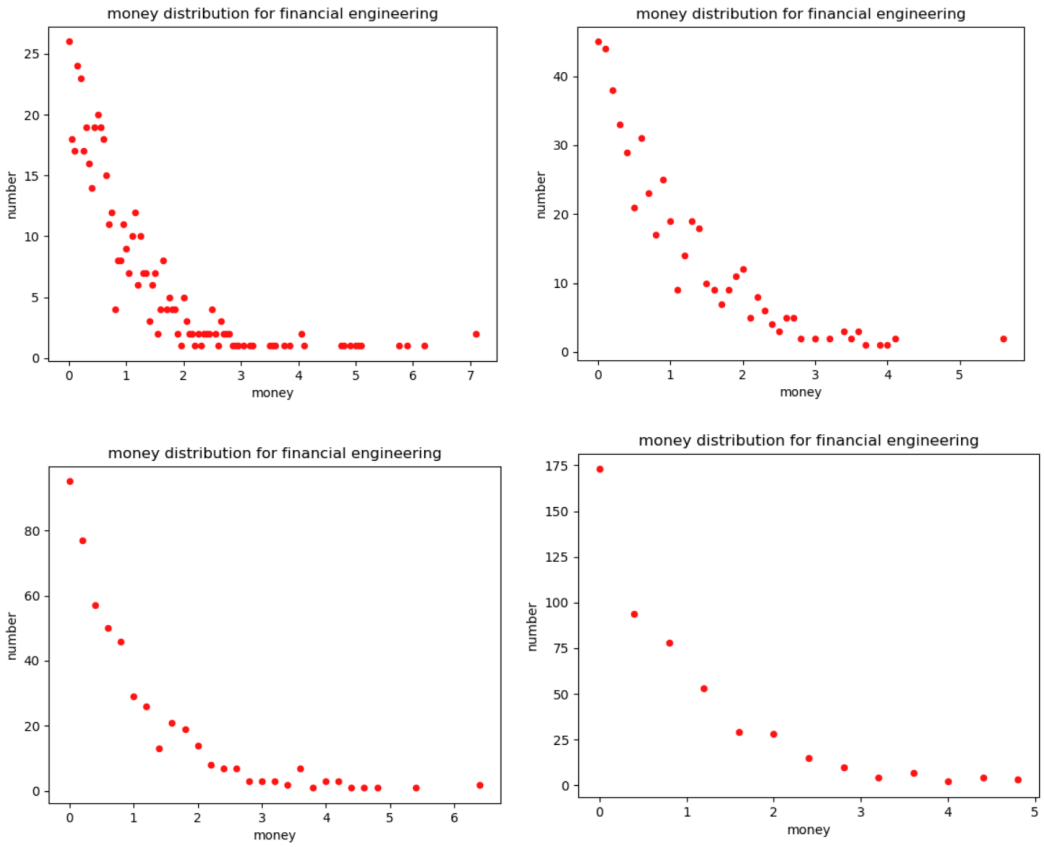



Figure 12　image of the money distribution when the step is at 0.05, 0.1, 0.2, and 0.4

It can be seen that the distribution of money has a clear form of exponential decline. When the transaction reaches a dynamic balance, most agents have less money, and only a small number of stock market agents have more money. As can be seen from the fourth figure, the capital m and the count number present the following form:

$$\omega_m = \beta exp(-\beta m)$$

## 5b

In this part we get the scatter image of log(wm) and m. The step required by the project is in the range of 0.01-0.05. After the experiment, it is found that the step of linear descending at step 0.4 is more clear and neat, the following figure is the log(wm)-m image with step at 0.05, 0.1, 0.2, and 0.4 respectively (from left to right, up to down):

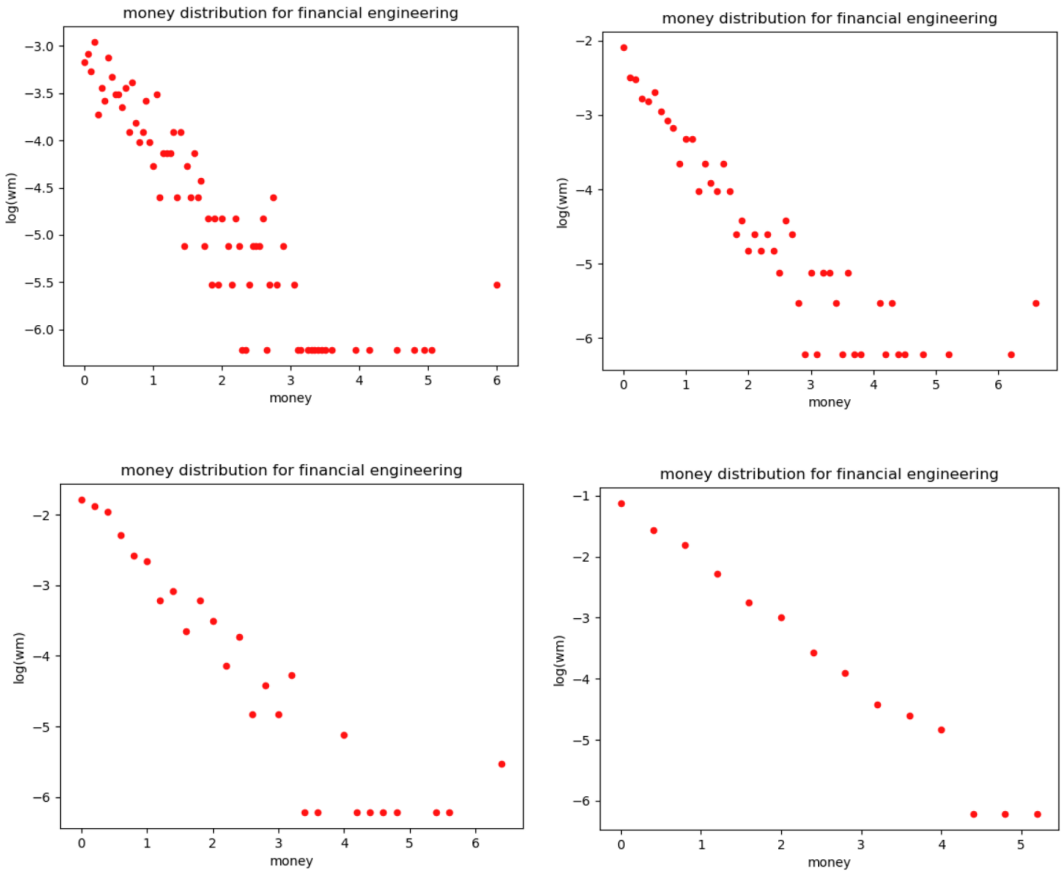



Figure 13 log(wm)-m image with step at 0.05, 0.1, 0.2, and 0.4

Because there are fewer agents samples with more money, it is more likely to have fluctuation, so you can ignore several points when money is large. As shown by the step 0.4, log(wm) and m are in a straight-down relationship. In line with the theoretical situation.

## 5c

In the task 5c we get the transaction money distribution image under different saving index z. The figure below is a scatter plot of the distribution of money when z is 0.25, 0.5 and 0.9 respectively. I set the step to two different values for comparison

When z=0.25, the steps are 0.05 and 0.3 respectively.

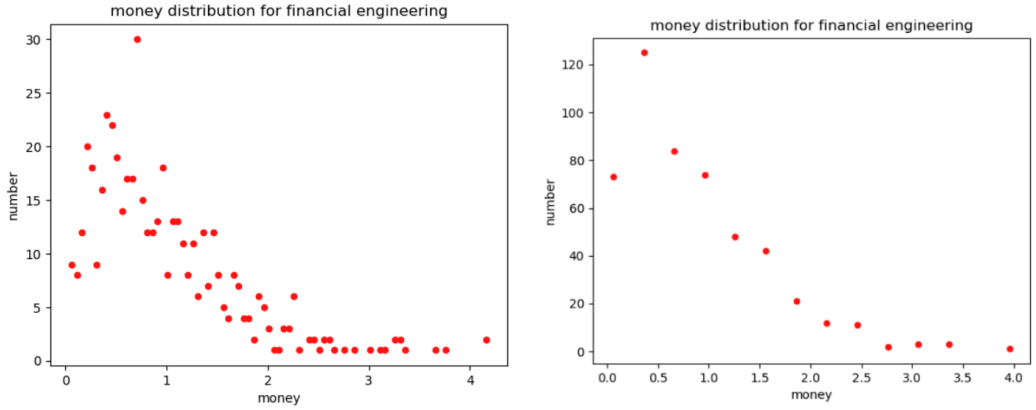



Figure 14 image when z=0.25, the steps are 0.05 and 0.3

When z=0. 5, step is 0.05 and 0.2 respectively

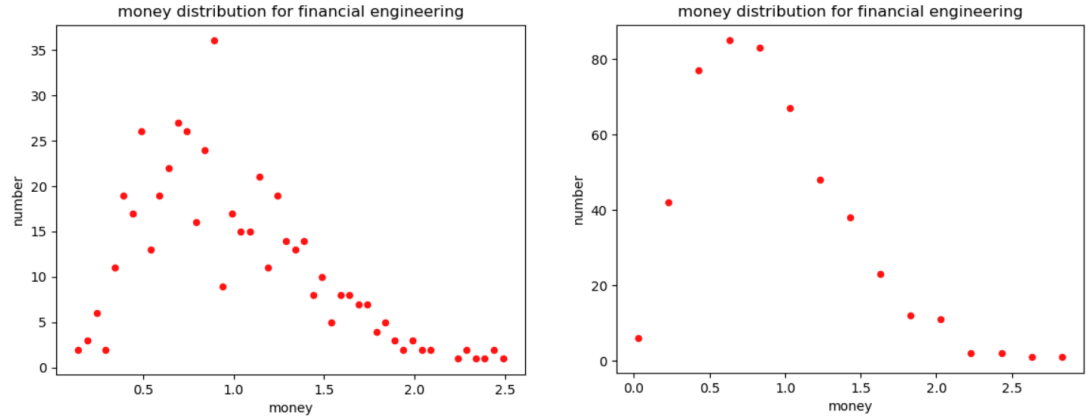



Figure 15 image when z=0.5, the steps are 0.05 and 0.2

When z=0.9, step is 0.05 and 0.1 respectively

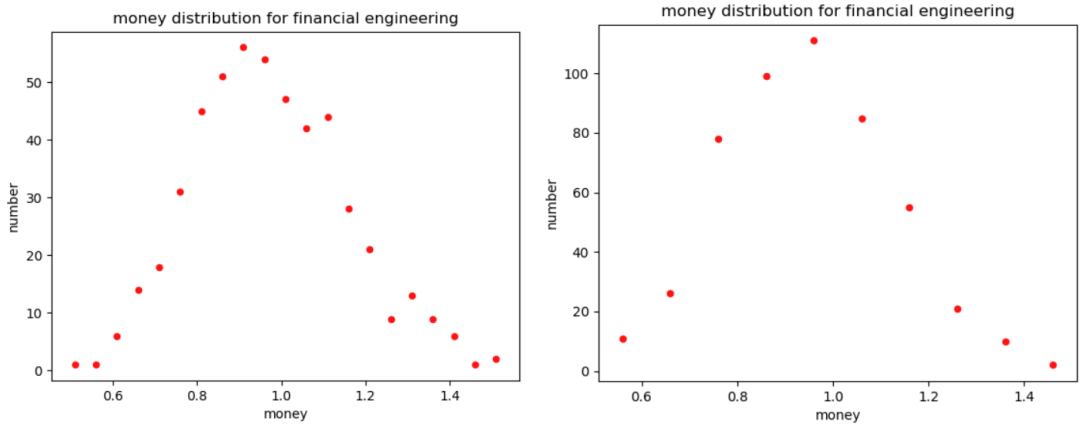



Figure 16 image when z=0.9, the steps are 0.05 and 0.1

It can be seen that the peaks of the scatter plot are basically at z*m.

## 5d

In this section we examine the nearest neighbor interactions. After compiling and running the program, we obtained the image of the money distribution at N=500 and 1000, with a= 0.5, 1.0, 1.5, and 2.0, respectively.

When N = 500, a is 0.5, 1.0, 1.5 and 2.0, respectively, it can be seen that the form of decline is roughly m^-1-a

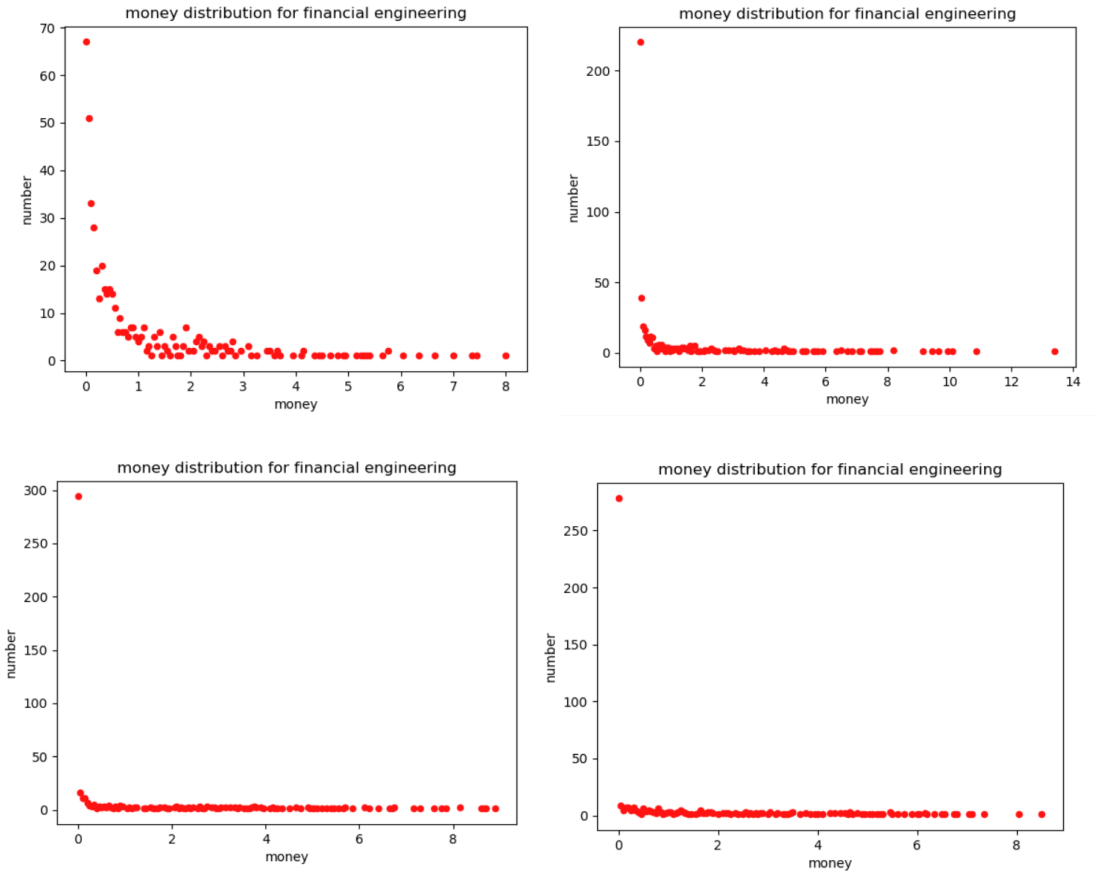



Figure 17 image for 5d when N = 500, a is 0.5, 1.0, 1.5 and 2.0

When N = 1000 and a is 0.5, 1.0, 1.5 and 2.0, respectively, it can be seen that it is a descending form of m^-1-a.

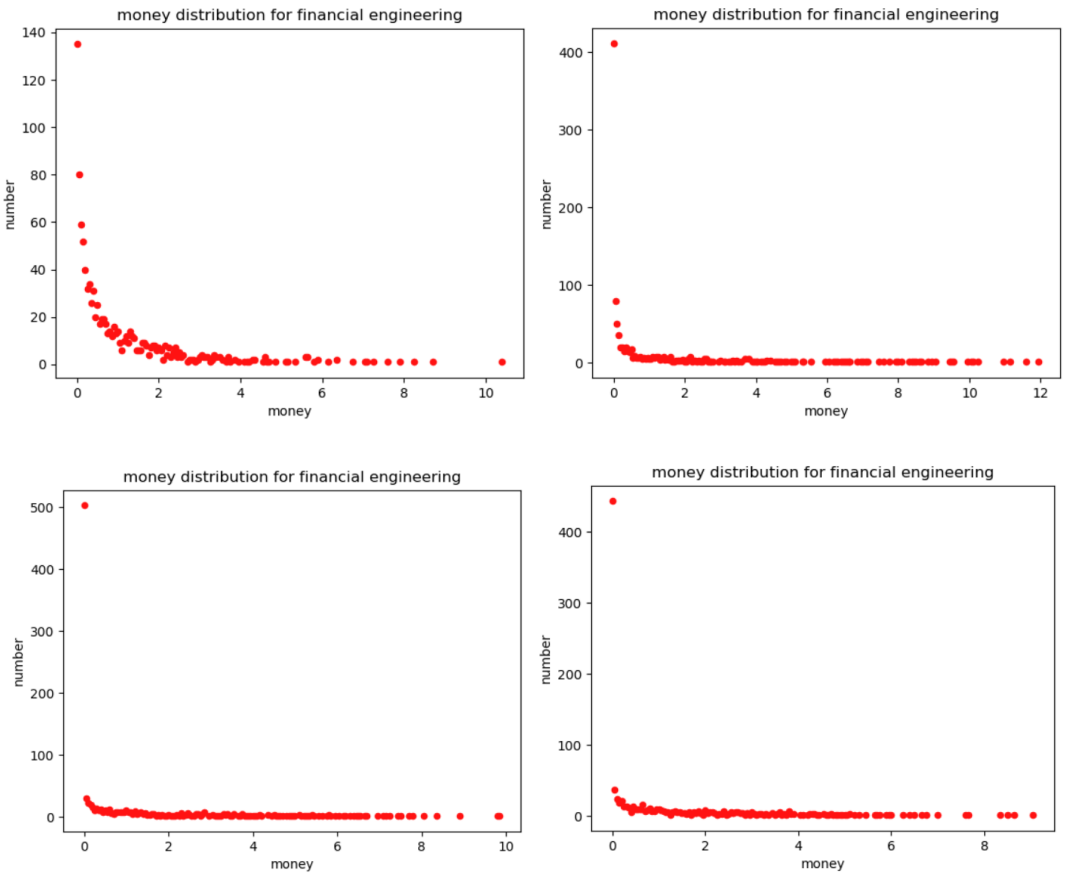
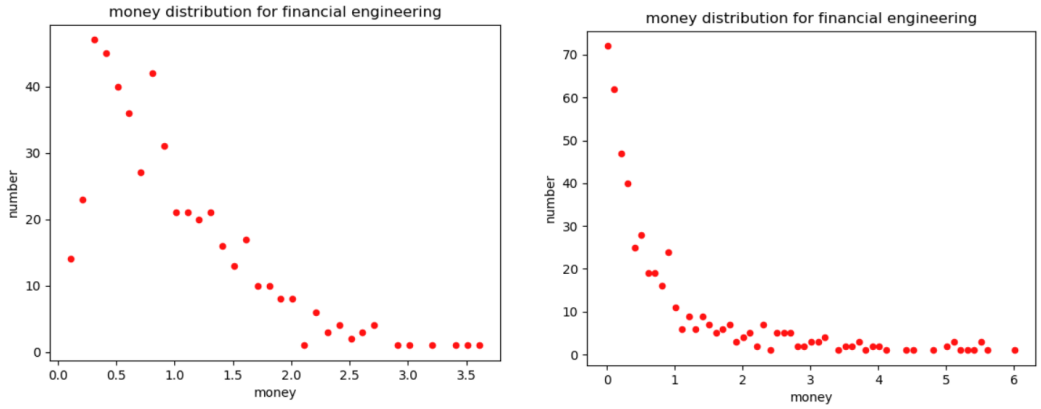



Figure 18 image for 5d when N = 1000, a is 0.5, 1.0, 1.5 and 2.0

After considering the addition of the saving index, the image of the money distribution will change. Taking the saving index as a typical z=0.5, the corresponding transaction image is obtained.

At z = 0.5, N = 500, a is an image at 0.5, 1.0, 1.5, and 2.0, respectively. It can be seen that there is a peak at around z*m=0.5.



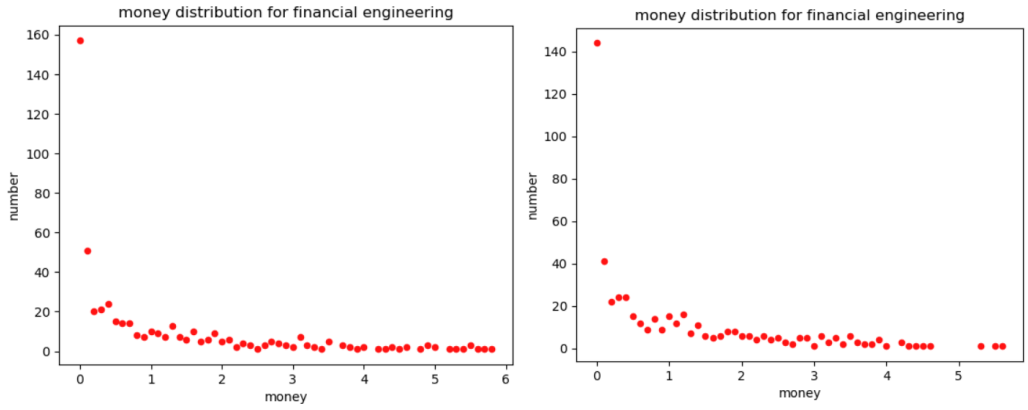



Figure 19 image for 5d when z=0.5, N = 500, a is 0.5, 1.0, 1.5 and 2.0

At z = 0.5, N = 1000, a is an image at 0.5, 1.0, 1.5, and 2.0, respectively. It can be seen that there is a peak at around z*m=0.5.

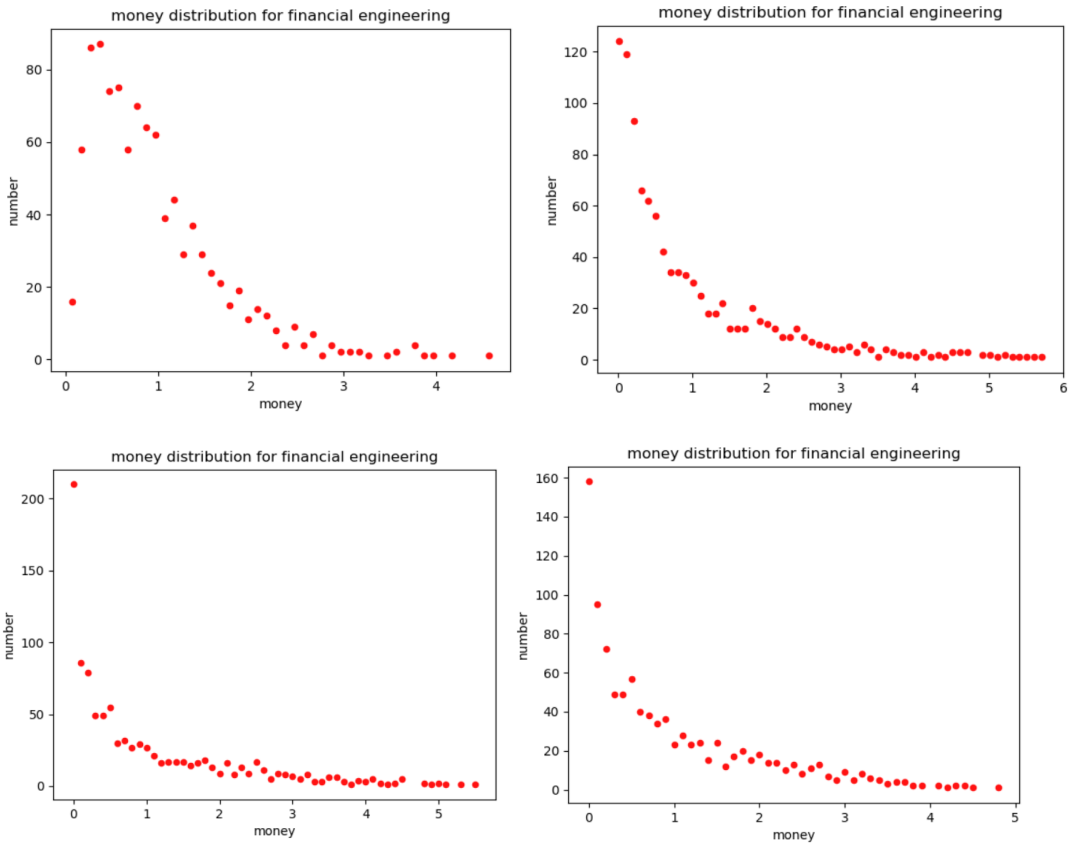



Figure 20 image for 5d when z=0.5, N = 1000, a is 0.5, 1.0, 1.5 and 2.0

## 5e

In this task, we also considered the former transaction influence. After introducing this item cij, we obtained images at N=1000, a=1.0 and 2.0, y=0.0, 1.0, 2.0, 3.0 and 4.0.

When a=1.0, y=0.0, 1.0, 2.0, 3.0 and 4.0

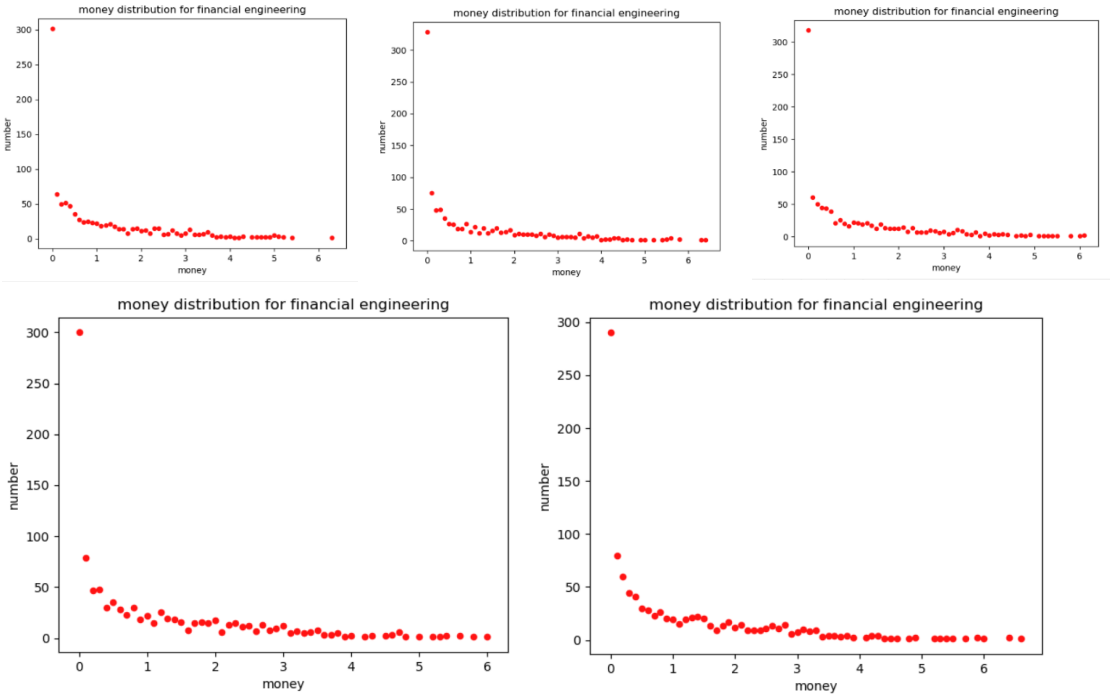



Figure 21 image for 5e when a=1.0, y=0.0, 1.0, 2.0, 3.0 and 4.0

When a=2.0, y=0.0, 1.0, 2.0, 3.0 and 4.0

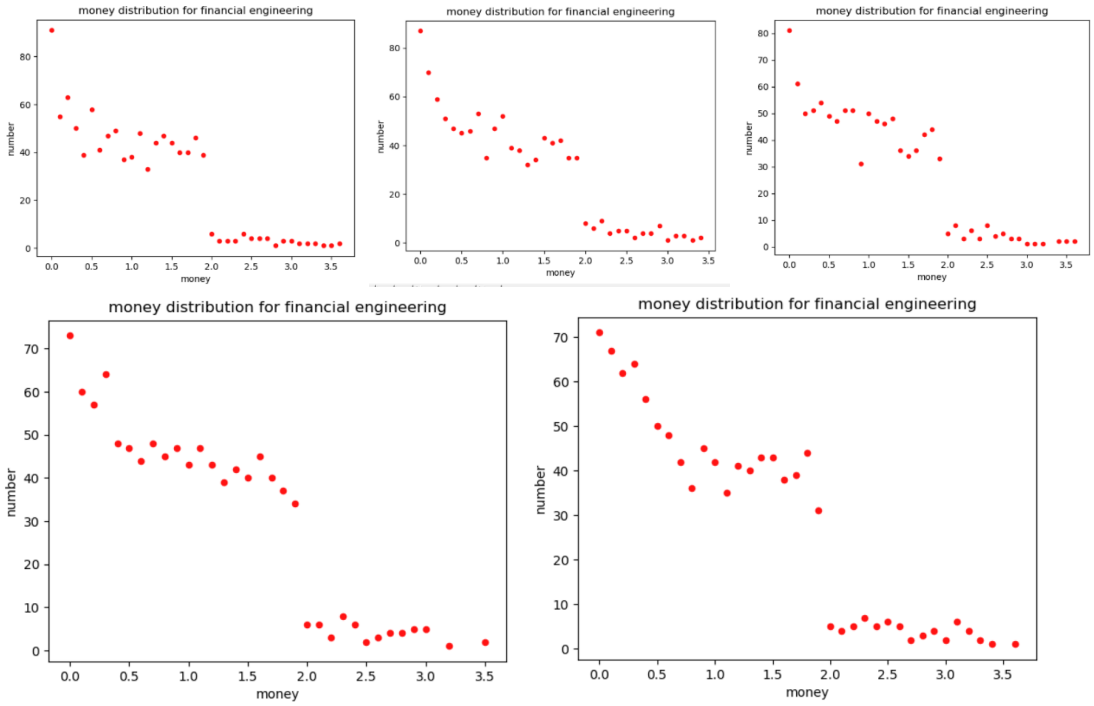



Figure 22 image for 5e when a=2.0, y=0.0, 1.0, 2.0, 3.0 and 4.0

As can be seen from the above figure, when a = 2, the image will have a certain cliff-like decline at roughly m = 1.8.

The above is the distribution of money without considering the saving, the code is added and rewritten, adding the saving index, setting z to the typical 0.5, re-simulating, we can get

money distribution at z=0.5, N=1000, a=1.0 and 2.0, Images of y=0.0, 1.0, 2.0, 3.0, and 4.0.

When z=0.5, a=1.0, y=0.0, 1.0, 2.0, 3.0 and 4.0

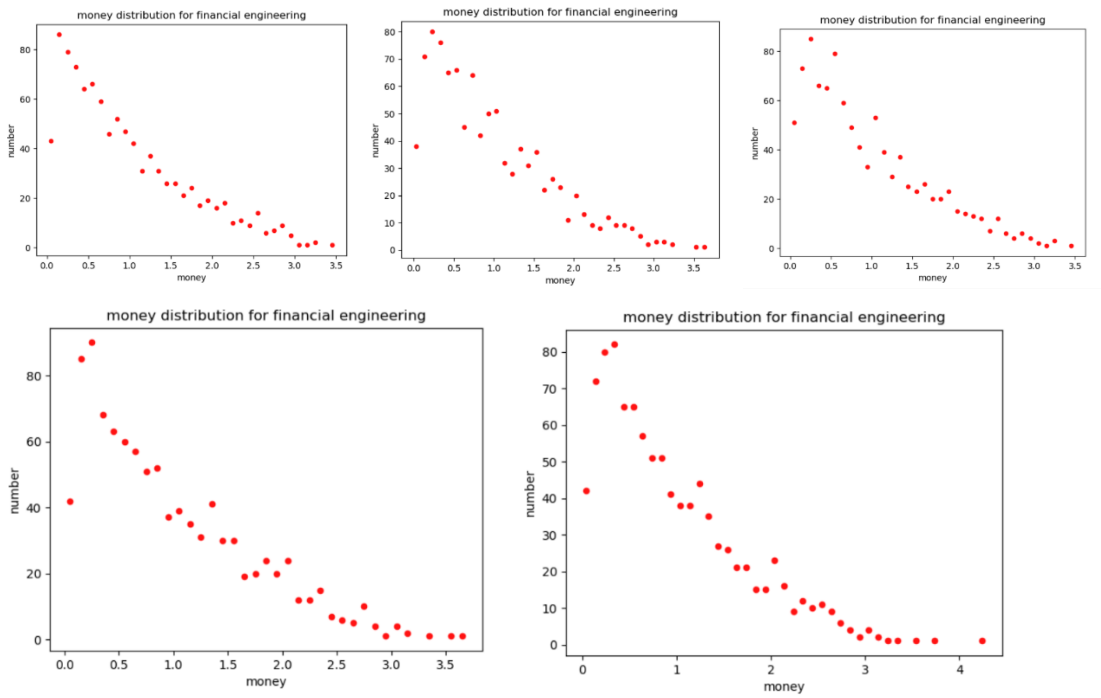



Figure 23 image for 5e when a=1.0, y=0.0, 1.0, 2.0, 3.0 and 4.0, with z=0.5

When z=0.5, a=2.0, y=0.0, 1.0, 2.0, 3.0 and 4.0

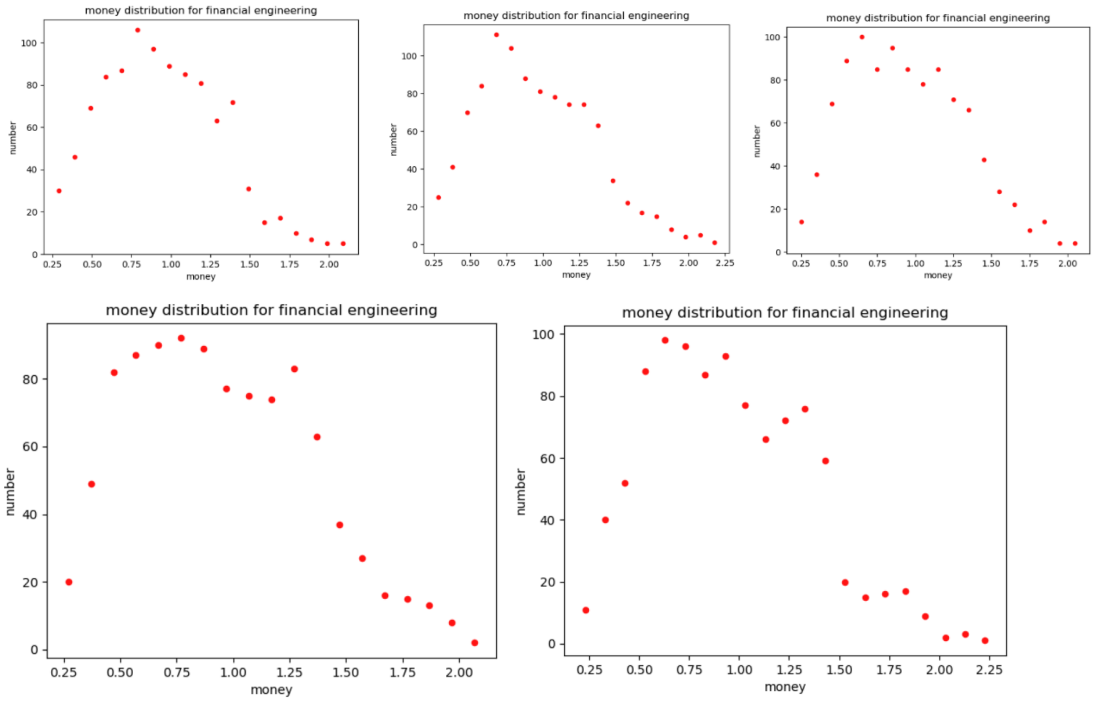



Figure 24 image for 5e when a=2.0, y=0.0, 1.0, 2.0, 3.0 and 4.0, with z=0.5

It can be seen that at a = 1.0, all different y have a peak around m = 0.25; and at a = 2.0, all different y have a peak around 0.75.

## 5. Concluding Remarks

This project has done a good job of simulating the stock market model, and successfully obtains the figure for money distribution after transaction.

The result is very well. We obtained the image of the money distribution under various assumptions and different indexs, and completed the requirements of each part of the project. The results obtained are also discussed. Due to Python's precision limitations and a small amount of error due to dynamic balancing, the resulting image will have some fluctuations, but it does not affect our ability to obtain correct results. Therefore, this program is reliable and convincible By changing the size of the step, we are able to get neater, clearer and more intuitive images. In general, although there is a certain limit in the speed and accuracy of the program, this program has successfully completed the simulation of the stock market model, met the requirements of the project, and obtained the corresponding images and conclusions. We can improve the data accuracy and speed of our program by using C++ simulation to improve and make it better for advance.

## 6. Reference

Hjort-Jensen,M., 2015. Computational physics.

The program can be found in https://github.com/lch172061365/Computational-Physics.git