An Overview of Agent-Based Modelling With Applications in Finance and Economics

Jahed Ullah 2139220

March 22, 2024

Supervisor: Junqi Tang

Abstract

Agent-Based Modelling (ABM) is an emerging and dynamic alternative to traditional modelling approaches. In this paper, we introduce the concept, detailing its origins, build a mathematical framework for Agent-Based Modelling, demonstrate simple examples and build a market model to show the effectiveness of ABM as a prediction tool for investors and economic policy makers.

Contents

1 Introduction

Modelling the behaviour of the stock market and financial institutions is a popular and lucrative field of study. There are various methods that have been developed over the years that attempt to model economical and financial behaviour, each with their own benefits and drawbacks. Computational Social Science is a field which comprises of some of these methods of modelling. Computational Social Science is the use of computers and computational methods to model social phenomena (Conte et al., 2012). Social phenomena can refer to a great number of things, both micro and macro, whether we want to look at the individual relationships between people or if we want to look at war or financial crisis'. There are various methods that are used in Computational Social Science. It combines techniques from mathematics, statistics and computer science to analyse and model social behaviour. One of the more prominent approaches is Agent-Based Modelling (ABM), which is the basis of this paper. ABM involves modelling systems and their entities as individual agents that interact with each other based upon some pre-defined rules. We will endeavour to explore ABM, and to demonstrate its usefulness in the field of finance and economics.

1.1 The History of ABM

ABM has been around for almost as long as computers. It began as an abstract modelling technique used by physicists in the 1930s, and it remained this way for some time. Large US universities set up big computing arrays to support scientists in the natural sciences, which allowed them to develop ABMs (Hanappi, 2017). Early work was done in the field of Cellular Automata (CA) by John von Neumann and Stainslaw Ulam. CA is a simpler form of ABM where the agents are cells on a grid which can take two states, we will explore this further in later sections. The popularity of CA began to grow once early researchers noticed that these simple models were able to produce emergent behaviour. Emergent behaviour is macro-level outcomes of a simulation caused by the micro-level interactions between the entities in the model. Notable mathematicians in the field of CA at the time include Wolfram and Langton who both worked on papers which created classifications for types of CAs (Heath and Hill, 2008).

In the 1970s and 1980s we began to see developments in computer technology, which allowed for the development of more complex simulation techniques. This, naturally, led to developments in the field of ABM as technology could now handle more complex simulations which allowed for the creation of models with considerably more advanced agent behaviours. Researchers began to experiment with multi-agent systems based upon the foundations laid by CA to study fields such as ecology, economics and social science. Early work on emergent behaviours such as cooperation and segregation models using ABM were done by the likes of Thomas Schelling and Robert Axelrod. The key idea behind a lot of this work was to demonstrate how complex patterns on the macro level could emerge from local interactions between agents (we will explore this in a bit more detail in later sections).

The 1990s and early 2000s can be viewed as a renaissance for the field of ABM. Spurred

by the development of highly competent computers and the development of dedicated, user-friendly software, applications of ABM began to become more common. In this era, we saw ABM being applied to a wide range of fields, most notably in economics, ecology, biology, sociology and political science. We also began to see the establishment of dedicated journals and conferences specifically for the discussion and research of ABM. This formalisation greatly contributed to the consolidation of ABM as a viable, mainstream modelling approach.

In the modern day, ABM is recognised and respected as a powerful simulation technique with applications in a range of industries. In the finance field, it is not uncommon to hear the use of ABM in the biggest financial institutions, including the likes of the Big 4 (KPMG, EY, PwC and Deloitte). The use of ABM can be seen in a large number of fields, notably finance, urban planning, traffic/transport, public health, biology, disaster response etc. Current research trends include the scalability of ABM and increasing realism of models. With the emergence of AI, we are on the cusp of a revolution in the field of ABM, as more and more advanced models and agents can now be built.

1.2 The Role of Computers and Artificial Intelligence

In the previous section, we mentioned how the emergence of AI will have a profound effect on ABM, we continue to discuss this in more detail in this section. Computational Social Science as a whole uses computers and computational methods as its foundation. Technological advancements are happening at an exponential rate. In 1965, Gordon E. Moore made an observation which came to be known as Moore's Law, which states that the number of transistors on microchips doubles every year whilst costs reduce. It also states that this growth is exponential. Thus far, Moore's Law has provided an accurate estimate of computer development. This is of interest to this paper due to the recent developments in Artificial Intelligence. Large Language Models (LLMs) such as OpenAI's ChatGPT and Google's Gemini have brought AI to the forefront of public interest. Through machine learning and continuous use from the public, these LLMs are regularly improving and providing key data for the companies behind them. Using Moore's Law, developments in the field of AI can be assumed to be exponential, with the near future AI being leagues ahead of what we have today.

All of this is important when discussing the future of ABM and Computational Social Science. Quantitative ABM has historically been less developed than qualitative ABM, however developments are being made in this field (Conte et al., 2012). The difficulties lie in analysing huge sets of data and properly modelling advanced and complex systems. AI can greatly improve ABM as a modelling technique as machine learning algorithms can help to extract and recognise patterns in large sets of data. AI can also improve agent behaviour. This is because it can allow the agents to learn over time and adapt their responses, which is much more realistic and can thus allow for more accurate models. Thus the developments of AI, which may grow exponentially when considering Moore's Law, will have a profound impact on the accuracy and applicability of ABM and Computational Social Science as a whole.

2 Agent-Based Modelling

Agent-Based Modelling (ABM) refers to a class of computational techniques that have proven useful as a way to model and represent the behaviour of individuals when studying social phenomena (Axtell and Farmer, 2022). They are essentially computer simulations of interactions between individual entities, which are known as agents. Agents are given an individual set of rules and behaviours and then are allowed to interact with each other. For example, if we were to model traffic in a city, each car would be an individual agent and they would be given a set of rules, such as "stay x distance from the car in front" or "stop at give-ways". They may also be given a destination and may want to optimise their route in order to minimise journey time. By observing how the individual agents (the cars) interact with each other, we can simulate and interpret how traffic patterns emerge. That is the overall idea of ABM. ABM has advantages over standard game theory and analytical modelling approaches, as the mathematical restrictions of these approaches cannot account for the heterogeneous nature of social phenomena, or look at behaviour outside of equilibria (Bianchi and Squazzoni, 2015). Here, heterogeneous means that each agent has their own unique characteristics, objectives and behaviours. ABM can account for heterogeneous social phenomena, and this is a reason why an ABM model can provide a more realistic output than traditional approaches.

ABM, in general, begins by specifying a population of agents, each with their own specific behaviours. In traditional economic models, it is common for agents to each be assigned a utility function, and then we can derive agent behaviour under the assumption that each agent would maximise their specific utility function. However, ABM takes a different approach which can be viewed as perhaps more realistic. In an ABM, agents can still be assigned utility functions which they may attempt to maximise however they may not succeed in doing so. The agents could also have an optimal response function which would maximise their function/outcome but, as may be the case in real life, they are permitted to maybe make errors. If we have a certain initial state S for our agents who engage in their specified behaviours B, then after some number of interactions between agents N, the agents will have a new state $S' = B(S, N)$ (Axtell and Farmer, 2022). Over time, one can study patterns of behaviour that emerge through repeated iterations of this process. We will discuss this further in later sections.

2.1 Key features of an ABM

Agent-based modelling can be conducted using a variety of software. In this paper, we will focus on approaches using Python, however all the various software (NetLogo, C++ etc.) will create an ABM with the same general features which we will state below, based upon Axtell and Farmer (2022). These features are:

- At least one population of defined agents which can represent individuals or groups (such as banks in the context of finance). This usually includes some information about the current state of the agents/population, such as their current profit etc.

- Defined agent behaviours which are heterogeneous (different for each agent) across the population, for each population if there is more than one. They are heteroge-

neous as the behaviour often depends on the state of the agent, which will vary from agent to agent.

- An external environment in which the agents are situated in, this could be aggregate economic variables, an interaction network or a spatial landscape.
- A way to statistically assess the state and behaviours of both the agents and the external environment, which is often then depicted visually so that the model can be assessed properly.

This is the general structure of an ABM, and this structure is left intentionally vague. It is through this vagueness that ABM excels, as it is applicable in various different fields by building models with these features.

2.2 Cellular Automata

ABM can be considered in three different formats. We have the simplest form, known as Cellular Automata, which we will be discussing in this section. We also have Classical ABM and AI based ABM, both of which we will discuss in later sections. Cellular Automata (CA) is the simplest form of an agent based model. As discussed earlier, it is the very foundation upon which ABM was built. CA works very similarly to the ABM models explained earlier and shares many of the same features as discussed in Section 2.1, however the models built here are much simpler. In CA, the agents are referred to as 'cells' and they are fixed onto a grid. Often the agents can only have 2 states, i.e. alive/dead or on/off etc. For this you can think of an OLED screen as an example to visualise what a CA model may look like, that is on an OLED screen the pixels are fixed, similar to our cells, and they can be either on (and coloured, but this would be a more advanced CA with multiple states) or off, and their state depends on their neighbours and the overall image of the screen. In some more advanced CA models, the cells can have more than 2 states however in these cases it is often preferable to use a traditional ABM. For Cellular Automata models, the rules of each cell describe its state at the next time step dependant upon the state of the cells in the neighbourhood of it.

Models of this form are often too simple and cannot really be used to describe true social relationships, as relationships are often more complex than just ones neighbours. Also, as mentioned earlier, a CA model with more than two states can be set up but it is often ineffective for modelling systems with many states. However, CA can be a useful tool to understand simple starting points of complex social behaviour. We will endeavour to create a simple Cellular Automata model using Python, and then demonstrate how we can analyse behaviour from the model. Our CA model can be seen in Section 2.6, however it is recommended to read through the following sections first, in order to build understanding prior to building the model.

2.3 Classic ABM And Simple Model

Traditional ABMs follow a similar format to that described earlier in this section. In a classical ABM model, we have a population of agents each with their own clearly defined

rules and each agent has its own distinct behaviours. Agents can perform actions within the model, but their actions must always fall into one of the following four categories (Hamill and Gilbert, 2015):

- **Perception** The agents can see each other in both their neighbourhood and environment as a whole
- **Performance** Agents can act with other agents and with their environment by moving or communicating
- **Memory** Agents can recall their previous states and actions and can learn from them, this will be particularly important when we discuss AI based ABM
- **Policy** Agents have pre-determined rules that dictate what they will do next.

This can all be demonstrated using a simple economic model, similar to the one used by (Hamill and Gilbert, 2015). For this model, we will imagine a simple shopping centre where all the shops sell the same category of products, for the sake of this example we will say various food and drink products. We have a population of shoppers, all with their own unique shopping lists, and the shoppers want to obtain every product on their list and they may even want to minimise their costs. Now, the shop keepers will sell the food and drink for differing prices, dependant on various factors such as what they assume shoppers will pay and sourcing costs from wholesalers. Now for this model it is important to note that not all of the shops sell the complete range of food and drink. We can model this on Python, however we want to first build a simpler model in the following sections. For now, we can show how this example can be modelled as an ABM. In this example, the agents would be both our class of shoppers and the shop keepers. Their environment is the shopping centre. The agent class of shoppers want to minimise the cost of their shopping so their behaviour would be of the form $minf$, where f is some function of the cost of their shopping. The shop keepers behaviour would be of the form $max\ C(x, y)$, where C is a function of their profit which takes the inputs x for income and y for expenses. This is how an ABM may look, we will use the concepts discussed here in later sections so that we can build our own models.

2.4 Finite Dynamical Systems

Traditionally, ABMs lack a true mathematical framework. This is both an advantage and a drawback of this form of modelling, that is because it is rather difficult to analyse the model mathematically as one would with a traditional model, however this also allows for more realistic models, as we know that entities in real life are likely to act based upon their neighbours and their own self-interest, rather than as a mathematical equation dictates. Nevertheless, we will endeavour to derive a mathematical framework for agent-based modelling. An idea for modelling ABMs mathematically can be as a time-discrete dynamical system on a finite set representing the possible states of the agents in the model. We will base this framework on the work by (Laubenbacher et al., 2007).

We will begin by defining the key variables and functions that will make up our mathematical framework for ABM. Let $x_1, ..., x_n \in F$. Here, the x_i represent the agents

that are being modelled, and F is a finite set of all the possible states the agents can take. Recall that in an earlier section, we discussed how agents are given *utility functions* which dictate their behaviour. We will define a similar set of functions for our mathematical framework. Each of our x_i, which represent the agents, are assigned *local update functions* $f_i : F^n \longrightarrow F$. Note that the input of f_i is F^n rather than F, this is the reason we refer to the functions as 'local' update functions. Here 'local' means that f_i takes inputs from the variables in the neighbourhood of x_i, we will show later that this is useful because agent behaviour is dependent on the states of their neighbours, as mentioned earlier. By abuse of notation, we also allow f_i to denote the function $F^n \longrightarrow F^n$, where the function changes the i^{th} coordinate whilst leaving the others unchanged. This allows for a sequential composition of the update functions (Laubenbacher et al., 2007). These functions combine to give us our dynamical system

$$\Omega = (f_1, ..., f_n) : F^n \longrightarrow F^n.$$

The dynamics of our system Ω can be described by two graphs. Our first directed graph is a mapping of elements in F^n to F^n. That is, $u \in X^n$ maps to $v \in F^n$ if and only if $\Omega(u) = v$. Our second graph relates to the fact that the states of our agents depend upon the states of other agents in their neighbourhood. That is, we have a graph between the vertices $V = 1, ..., n$, where there exists an edge from i to j if and only if x_i appears in the function f_j. That is to say, if the local update function for j depends upon x_i, then there is an edge between the two. The second graph is directed in general, but in practice this is not the case. If we consider the context of ABM, if agent x is in the neighbourhood of agent y, it is often the case that agent y is in the neighbourhood of agent x. Therefore, the second graph can be seen as symmetrical (Laubenbacher et al., 2007). This means it is often the case, variable x_i appears in $f_j \iff x_j$ appears in f_i. Therefore, in this case the second graph can be thought of as an undirected graph.

2.4.1 Stochastic Dynamical Systems

The framework we have laid out above can be made stochastic. This can prove to be particularly useful in the context of financial mathematics, where Dynamic Stochastic Modelling is common. There a number of ways that this can be done, as laid out by (Laubenbacher et al., 2007). The first is assuming that each of our variables x_i are assigned a set ($\neq \emptyset$) as well as a probability distribution over this set. At each new time-step, a local update function is chosen at random from the set, this is known as a *probabilistic finite dynamical system* (PFDS). The other approach involves using a set of permutations $J \subseteq S_n$, where S_n is the set of all possible permutations of n objects, and a probability distribution. At each time-step, a permutation $\pi \in J$ is selected at random and used to update every agent, sequentially, this is known as a *stochastic finite dynamical system* (SFDS).

We would like to describe the stochastic phase space of our dynamical system using the following. Let $\Lambda = \{\Omega_1, \Omega_2, ..., \Omega_t\}$ be a finite collection of systems and let $p_1, ..., p_t$ be a collection of probabilities such that they all sum to 1. Then our stochastic phase space is

$$\beta_\Lambda = p_1\beta_1 + p_2\beta_2 + ... + p_t\beta_t,$$

where each β_i is the respective phase space of Ω_i. Then our stochastic phase space β_Λ can be viewed as a Markov Chain over F^n.

2.5 Agent Based Models as Finite Dynamical Systems

We now discuss how we can present ABMs a finite dynamical system. As stated earlier, the x_i represent our agents and each agent is assigned a local update function f_i which depends upon the state of the agents neighbours, i.e. on the other x_i. Our dynamical system Ω represents our model, as it is an amalgamation of all the local update functions. Sometimes, randomness may be preferred to be introduced into the model. For example, if we are studying financial markets or cell division, we require a degree of randomness. In this case, our framework can be viewed in a stochastic way as described by the previous section. Here the Ω_i represent the set of update functions that agent x_i can take, and through the use of probability and random variables we build a stochastic model where the specific update function of each agent is randomised at each time step. So one may wonder now, what is the benefit of this framework? Well, this framework provides guidance for how one may analyse an ABM mathematically. Since we have now detailed how an ABM can be represented as a finite dynamical system, analysis can be conducted on the system using traditional mathematical methods from the field of finite dynamical systems. We will not detail which methods these are or how they can be conducted as that is beyond the scope of this paper. In this paper, we just want to introduce how ABM can be viewed mathematically and we provide a framework to do so.

2.6 Conway's Game of Life

We can demonstrate an example of an ABM, specifically a cellular automation, as a finite dynamical system using Conway's Game of Life. Developed by John Conway in the 1970s (Vayadande et al., 2022), the game is based on an infinite grid (but often finite for computational purposes) with cells that can take two states, either 'Alive' or 'Dead', and the cells are updated at each time step depending on a number of rules which depend on the cells in the neighbourhood of it. Whilst rather simple, Conway's Game of Life has useful properties for studying emergent behaviour.

We can define the rules of Conway's Game of Life mathematically. Here, the x_{ij} are the cells, which update based upon local update functions which we will define. Suppose we have a Cartesian grid as our space, where the cell at point (i, j) is denoted as x_{ij}^t where t is the specific time step we are currently at. Let us denote the number of living neighbours in the neighbourhood of x_{ij} at time t as N_{ij}^t. We convert the states of 'alive' and 'dead' into the binary values 1 and 0 respectively. Then we have the following rules:

- **Birth Rule:** If we have a dead cell, that is $x_{ij}^t = 0$, then at the following time step it can become alive if it has exactly three live neighbours. That is to say

$$x_{ij}^{t+1} = \begin{cases} 1, \text{ if } S_{ij} = 0 \text{ and } N_{ij}^t = 3, \\ 0, \text{ otherwise.} \end{cases} \tag{1}$$

- **Survival Rule:** If we have a live cell ($x_{ij}^t = 1$), then it can remain alive if and only if it has 2 or 3 live neighbours. That is to say

$$x_{ij}^{t+1} = \begin{cases} 1, \text{ if } 2 \leq N_{ij}^t \leq 3, \\ x_{ij}^t, \text{ otherwise.} \end{cases} \tag{2}$$

- **Death Rule:** A live cell can die due to two factors, that is under-population or overpopulation. That is if a cell has fewer than 2 live neighbours or greater than 3 live neighbours then it dies. Mathematically, that is

$$x_{ij}^{t+1} = \begin{cases} 0, \text{ if } N_{ij}^t \in [0,1] \cup [4, +\text{inf}), \\ x_{ij}^t, \text{ otherwise.} \end{cases} \tag{3}$$

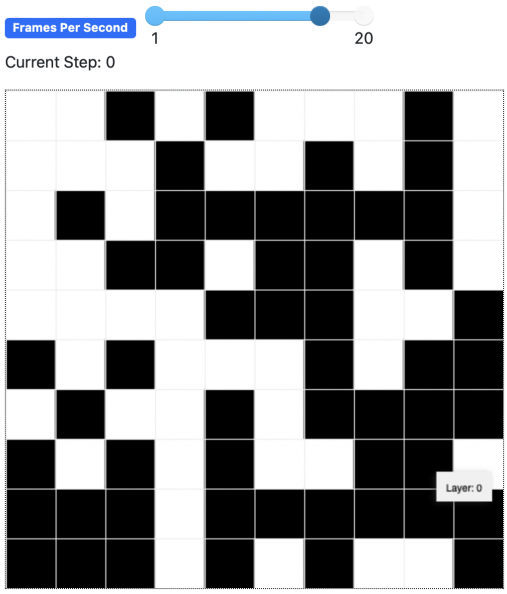
By applying these rules at each time step $t+1, t+2, ...$ we can observe the behaviour of the cells and study its outcome. We can model this in Python using the following code:

```
from mesa import Agent, Model
from mesa.space import MultiGrid
from mesa.time import SimultaneousActivation
from mesa.visualization.modules import CanvasGrid
from mesa.visualization.ModularVisualization import ModularServer

class Cell(Agent):
    def __init__(self, unique_id, model, x, y, initial_state):
        super().__init__(unique_id, model)
        self.x = x
        self.y = y
        self.state = initial_state
        self.next_state = None

    def step(self):
        self.state = self.next_state

class GameOfLife(Model):
    def __init__(self, height, width):
        self.height = height
        self.width = width
        self.grid = MultiGrid(width, height, True)
        self.schedule = SimultaneousActivation(self)

        # Create cells and place them on the grid
        for x in range(self.width):
            for y in range(self.height):
                cell = Cell((x, y), self, x, y, self.random.choice([0, 1]))
                self.grid.place_agent(cell, (x, y))
                self.schedule.add(cell)

    def step(self):
        for cell in self.schedule.agents:
            neighbors = self.grid.get_neighbors((cell.x, cell.y), moore=
    True)
            live_neighbors = sum(1 for neighbor in neighbors if
                                 neighbor.state == 1)
```

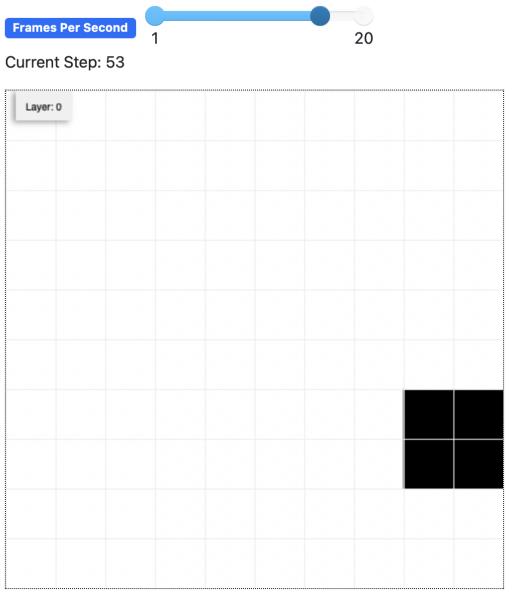
```
            # Implement the rules of Conway's Game of Life
            if cell.state == 0 and live_neighbors == 3:
                cell.next_state = 1
            elif cell.state == 1 and (live_neighbors < 2 or
                                      live_neighbors > 3):
                cell.next_state = 0
            else:
                cell.next_state = cell.state

        # Update cell states simultaneously
        for cell in self.schedule.agents:
            cell.step()

def agent_portrayal(agent):
    portrayal = {"Shape": "rect", "w": 1, "h": 1, "Filled": "true",
                 "Layer": 0}
    portrayal["x"] = agent.x
    portrayal["y"] = agent.y
    if agent.state == 1:
        portrayal["Color"] = "black"
    else:
        portrayal["Color"] = "white"
    return portrayal

grid = CanvasGrid(agent_portrayal, 10, 10, 500, 500)
server = ModularServer(GameOfLife, [grid], "Game of Life", {"height": 10,
                                                            "width": 10})
server.port = 8522  # The default
server.launch()
```

To do this, we have used a Python package known as Mesa (Kazil et al., 2020). This package was developed to allow users to easily and intuitively develop agent based models, as we have done above.

Let us discuss what is happening in the code. After importing the necessary packages, we begin by defining a set of agents and by defining a grid for the model to be visualised on. We then implement the set of rules for Conway's Game of Life, where we have assigned the binary variable 'cell.state' as 0 to indicate a dead cell and 1 to indicate an alive cell. After this, we allow the cells to be updated simultaneously at each time step. All of this combines to give us the following outputs:

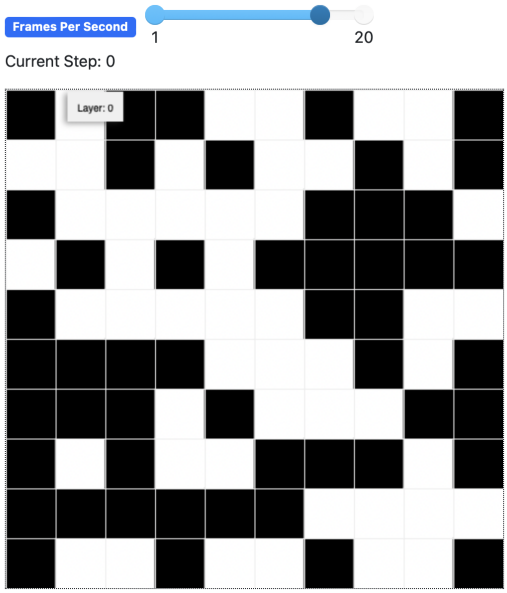


At time step 0

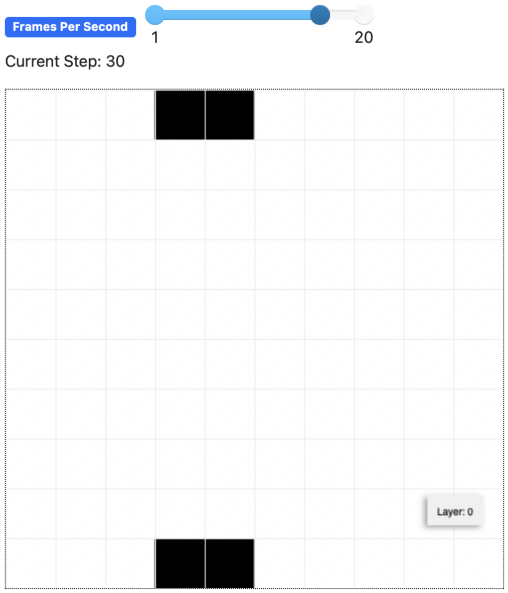


At time step 53, after equilibrium

Figure 1: One example of running our simulation



At time step 0



At time step 30, after equilibrium

Figure 2: Another example of running our simulation, with a different starting point

As you can see from our images, we have created a grid of black and white cells, where the black cells are alive and the white cells are dead. This is a form of Agent Based Modelling, more specifically a form of Cellular Automation. At each time step, the cells behave according to the specified rules, birthing new alive cells and other cells dying off. Whilst this looks incredibly simple, we can study a great number of phenomena from this simple model. For example one thing to note that can be seen in both figures,

after a number of time steps (which depends upon our starting configuration) we will always reach equilibrium/a steady state. This is a state where no matter how many time steps are taken beyond this point, the configuration will no longer change. In Figure 1, this time step is 53 and in Figure 2, this time step is 30. Despite the simplicity of the model, we can start to analyse emergent behaviour. However, when looking at emergent behaviour regarding ideas like cooperation and social norms, it may be better to look at a more complex model which we will look at later in this paper.

2.7 Emergent Behaviour

Emergent behaviours are overall, usually unexpected outcomes of an ABM caused by interactions between agents. These micro-level interactions can often lead to macro-level outcomes of the system. These outcomes are important as it often dictates the outcome of the ABM. For example, if were to create an ABM like the one we detailed in Section 2.3, we may be interested to see, for example, how the interactions between the shop keepers would impact the overall profits of each of them. Do they work together to ensure a stable and fair market, or do they undercut each other's prices leading to lower returns overall?

There are a number of different emergent outcomes that we may see in a model, some key examples are: *cooperation* where agents work together in different ways to achieve a shared goal; *social norms* where agents can conform to what their neighbours are doing; *social punishment* where agents may perceive other agents in a bad light due to previous actions and thus choosing not to interact with said agents, and many more types of emergent outcomes which can be detailed in (Bianchi and Squazzoni, 2015). We can actually see the formations of cooperation in our Game of Life simulation in Section 2.6, in Figure 1 the band of 4 cells have aligned their states so that they remain alive forever and the other cells remain dead, and similarly in Figure 2 the two alive cells at the bottom are co-existing as are the two cells at the top. These cells have noticed that through cooperating, they can manipulate their behaviour equations in order to remain alive and thus we reach an equilibrium in our model. Game Theory is often used to investigate cooperation in ABM, as the fundamental concepts behind it allow us to analyse our models as games of multiple players where Nash Equilibria can occur, so each player reaches an optimal response based upon the optimal responses of their neighbours. Essentially, it's a best case scenario situation. There are entire studies about Game Theory and how it relates to the outcomes of ABMs. This therefore shows the depth of the field of ABM as it is able to combine computational modelling, social science and the traditional mathematical field of Game Theory.

3 Agent-Based Modelling in Economics

Agent-Based Modelling has emerged as a viable and useful tool to model economic situations. Whether that be simple market models or complex models with numerous agents, each representing a firm or private investor, ABM has demonstrated its usefulness over other, more traditional economic models. ABM is a fairly recent approach, having only risen to popularity since the 1980s, and now with the increased development of computers

and recent developments in AI, ABM has positioned itself as a valid alternative to traditional economic modelling methods such as Dynamic Stochastic General Equilibrium models, for example.

Traditional models are often limited by a number of factors. One of these such factors is the common assumption of homogeneity of the agents. Traditional models often simplify the diverse behaviours and strategies of the agents, which is often unrealistic in a real life scenario. ABM allows for the agents to be heterogeneous, each with their own strategies and behaviours, which provides a much more realistic model. Another advantage of ABM is that, in more complex models, the agents can be programmed to have dynamic learning abilities, which allows them to adapt their strategies based upon their past actions. This is much more realistic than traditional models where agents are assumed to be static, or too rational. Therefore, in this section, we will use the theory discussed in Section 2 to develop our own ABM economic model and demonstrate how it can be used in the industry.

3.1 Motivation

When building a financial model, one will consider many factors during the process. For example, one would assume that investors would like to optimise their investment strategies, markets may be assumed to be perfectly efficient etc. When investors look at optimising their investment strategies, they will look at various pieces of data. One of the most common pieces of analysis they conduct is 'back testing' which is when investors look at the previous market prices of an asset in order to determine their return. However, there is an important piece of information to consider which back testing is unable to account for, and this is the 'market impact'.

Definition 1. *The **Market Impact** is when the price of an asset changes caused by the trading of the asset itself.*

What we mean here is that an investor trading an asset can themselves move the price of said asset. Assume an investor wants to sell shares of asset x, then if they sells a large enough quantity of asset x, they will actually drive the price of the asset down. This can be seen in the world of finance, where investors who are looking to sell/buy large quantities of shares will actually go to Asset/Wealth Managers who will then strategically sell or buy the asset so as to minimise the affect of the market impact. In traditional models, there is no way to reliably estimate the market impact before trading, therefore investors cannot reliably estimate their profits prior to trading. Furthermore, as the market impact changes the price of an asset, this itself would change the behaviours of other investors which further affects the price of the asset (Mizuta et al., 2022).

3.2 Building The Model

We will now endeavour to build an artificial market model to demonstrate how ABM can be used to model the price and volatility of a stock, and to determine if ABM is an effective approach for investors to optimise their trading strategies with regards to the market impact. Our model will be based upon the work by (Mizuta et al., 2022). We will

be describing the model using the terminology introduced in the earlier sections where we discussed a mathematical framework for ABM.

In our model, we will have a number of different classes of agents. These agents are two different technical agents which will use more advanced technical analysis techniques to motivate their investment strategies. The two technical are one momentum agent (TA-m) and one reversal agent (TA-r). What this means exactly will be explained further in the coming sections where we describe the behaviours and update functions of the agents. We will also include two classes of normal agents (NA), each class will have $\frac{n}{2}$ agents for a total of n normal agents, who do not follow as strict/advanced of a trading strategy as the technical agents, their decisions can be influenced by a mix of strategies, external factors, randomness and other agents. So, using our earlier terminology, our variables $x_1, ..., x_n$ are the classes of normal agents.

Our model will contain only one stock. We will investigate how the agents and their behaviours, and their inclusion in the model, affects the price of the stock. Similar to the paper by (Mizuta et al., 2022), we will use a continuous double auction to determine the market price of the stock.

Definition 2. *A double auction market is when a buyer's price and a seller's asking price match, and the trade proceeds at that price. (Tarver, 2021).*

3.3 Normal Agents

We will now introduce the behaviours of the agents in our model. We begin with our n normal agents. The normal agents will always place an order for exactly one share of the stock. When a normal agent orders a share of the stock, our 'tick time' t increases by one. The normal agents in our model represent general investors in the market, and they conduct trades using a mixture of fundamental and technical analysis strategies. In the paper that we are basing our model off of, (Mizuta et al., 2022), there is only one class of normal agents. In our model, we will have two different classes of normal agents with a slight difference between them. The second class of normal agents represent a slightly more seasoned investor, but not as advanced as professional investors which are represented by the TAs. Mathematically, there is only a slight difference between the two classes, which will be explained shortly. We now introduce a number of rules to dictate the behaviour of our NAs:

- Our NAs can short sell freely.
- The number of positions any given NA can hold is not limited, thus they can take an infinite number of positions, which can be either long or short.
- Each NA places an order to buy or sell the asset sequentially, this means that at $t = 1$ NA number 1 places their order and at $t = 2$ NA number 2 places their order etc.

To determine the expected return of agent j for $j = 1, 2, ..., n$ at time t, we use the following equation, as introduced by (Mizuta et al., 2022):

$$r^t_{e,j} = \frac{(w_{1,j} ln \frac{P_f}{P^{t-1}} + w_{2,j} ln \frac{P^{t-1}}{P^{t-\tau_j-1}} + w_{3,j\epsilon^t_j})}{\Sigma^3_{i=1} w_{i,j}}. \tag{4}$$

Here, $w_{i,j}$ is a weight term that is assigned to term i in the equation and can be unique for each agent j, it represents the importance that each agent places on the individual parts of the equation when trying to determine their return. P_f is a fundamental value and is a constant in the model and it represents the underlying or intrinsic value that the agents believe that the asset has. P^{t-1} is the value of the asset at the previous time step, and it is used by the agents as a form of technical analysis to estimate their returns based on previous market prices, this is similar to the back testing approach we mentioned earlier. $P^{t-\tau_j-1}$ is the price of the asset at another previous time step, specifically τ_j time steps before $t-1$, here τ_j is a parameter that determines how far back agent j will look for their asset price and it can vary from agent to agent. It again determines the individual technical analysis approach for each agent, where some agents will look further back in time to optimise their strategy. ϵ^t_j represents a noise or random shock variable and introduces an element of unpredictability to the market to increase the realism of the model, it is determined by random variables with a normal distribution, specifically $N(0, \sigma_\epsilon)$. Similarly, for our model, the τ_js and the $w_{i,j}$s are determined by random variables which are uniformly distributed on the interval $(0, \tau_{max})$ and $(0, w_{i,max})$, respectively, where τ_{max} and $w_{i,max}$ are values we state before running the simulation.

Now, we explain what each term of the equation actually represents in our market model. The first term represents the fundamental analysis process of our agents, the agents expect a positive return on their investment if the assets intrinsic value is higher than the market price, and they expect a negative return on their investment when the intrinsic value is lower than the market price. The second term is the technical analysis portion of the equation using a historical price of the asset, how far back they look varies from agent to agent, and the agents expect a profit when the historical market return is positive, and negative return if it is negative. Finally, the third term is the noise term and it introduces the element of unpredictability to the market to increase realism. This is because markets in real life are inherently unpredictable, they depend on unforeseen external factors such as news events or inflation.

The weight each individual agent puts on each part of the equation varies and the weights themselves actually creates the split in the classes of normal agents. The first class of agents, like the model in the original paper, have a fixed weighting for the duration of the model. The second class of normal agents can actually learn over time and adjust their weightings, much like a seasoned investor would learn based on previous trades in real life. The learning mechanism itself is based upon (Yagi et al., 2023). To explain the learning mechanism, we must first discuss how the NAs predict the price of the market at each time step. The expected price that NA j (for $j \in [1, ..., n]$) is calculated by:

$$P^t_{e,j} = P^{t-1} \times \exp(r^t_{e,j}), \tag{5}$$

that is, the price of the asset at the previous time step multiplied by the expected return for agent j at time t. We also now introduce the order price, this is the price that an agent is willing to buy or sell the asset. This depends on the agent, as in real life investors would have differing opinions on how much they believe the asset is worth. The order price at time t for agent j is:

$$P^t_{o,j} = P^t_{e,j} + P_d(2\rho^t_j - 1), \tag{6}$$

where P_d is a constant value that represents the depth of the market or a scaling factor for the distribution of order prices. It can be interpreted as the typical price deviation that agents are willing to accept from their expected price. ρ^t_j is determined by a random variable uniformly distributed on the interval $(0, 1)$ at time t, it introduces variability to the order price which simulates the scattering of order prices in the real market. Thus the order prices are scattered around the expected price which simulates the spread of limit order prices in a real stock market. Our model therefore accounts for the fact that not all investors will not want to buy or sell for the same price, which simulates the diverse opinions on the true value of the asset which exists in a real market.

Now that we have introduced these concepts, we can discuss the learning process of our second class of NAs. Using our earlier notation, if there are s of the second class of NAs (which will now referred to as NA2, for simplicity), then they would be denoted as $x_{n-s}, x_{n-s+1}, ..., x_n$. The learning process allows the NA2 agents to adjust the weightings in equation 4, through experience in the market gained by their previous trades. The process is performed before the order process at each time period. Let us define

$$r^t_l = ln\left(\frac{P^t}{P^{t-t_l}}\right),$$

this represents the technical analysis of the system as we are looking at the price of asset at time t compared to the price of the asset at time $t-t_l$, where t_l is a constant evaluation term which we state before building the model. Then the learning process for the NA2s is as follows:

$$w^t_{i,j} = \begin{cases} w^{t-1}_{i,j} - kr^l_t u^t_j w^{t-1}_{i,j} \text{ if } r^t_{i,j} \text{ and } r^t_l \text{ have opposite signs} \\ w^{t-1}_{i,j} + kr^l_t u^t_j (w_{i,max} - w^{t-1}_{i,j}) \text{ if } r^t_{i,j} \text{ and } r^t_l \text{ have the same sign,} \end{cases} \tag{7}$$

where k is a constant which we provide prior to running the simulation, u^t_j is a random variable that is uniformly distributed on the interval (0, 1) for each agent j. Thus we can see that the update process is no more than comparing the technical analysis of the system to the return of the individual agent and adjusting accordingly. Therefore, over time the agent will learn better parameters and switch to the investment strategy that estimates correctly, the fundamental strategy or the technical strategy (Mizuta et al., 2013).

Prior to running any simulations, it is important to note how the agents will determine whether to buy or sell the asset. This is all based upon the order price $P^t_{o,i}$ and the expected return of the agents $P^t_{e,i}$. Specifically, if $P^t_{e,i} > P^t_{o,i}$ then the agent will place an order to buy one share of the stock. If $P^t_{e,i} < P^t_{o,i}$, then the agent sells one share of the stock. If they are equal, the agent does nothing.

3.4 Baseline Model

We will now build and run our market simulation to establish some baselines. These baseline models will omit certain classes of agents to demonstrate their effect on the market. We will begin by running a simulation with only normal agents, specifically just the NAs who do not have a learning process to adjust their weights. To build and run our model, we will be using the Python package Mesa in a similar way as we have done in Section 2.6, however this model will be considerably larger and more advanced. We will place the full code in the appendix for reference.

In our initial baseline model, we set the following parameters: $n = 1000$, $w_{1,max} = 1$, $w_{2,max} = 100$, $w_{3,max} = 1$, $\tau_{max} = 100$, $\sigma_\epsilon = 0.03$, $P_d = 1000, P_f = 10000$, $t = 5000$. These parameters can all be individually adjusted as required by the model, in fact one may want to adjust each parameter whilst keeping all others constant in order to study the effect of each parameter on the market model. A key parameter that one may experiment with is P_f, which represents the fundamental price that the investors believe the asset has. In this initial baseline model, we have set this as equal to the initial market price, this means that the agents believe that the asset is priced correctly for its worth. However adjusting P_f would make the asset either undervalued or overvalued, which may change the behaviour of the agents. Upon running this simulation three different times, we receive the following outputs:

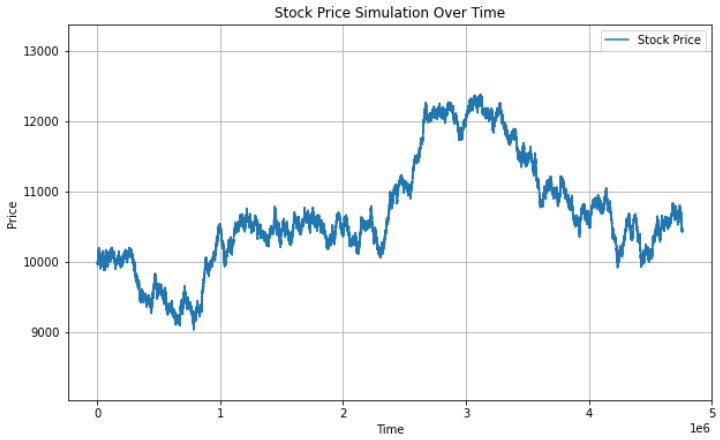


Figure 3: Run 1 of our baseline model.

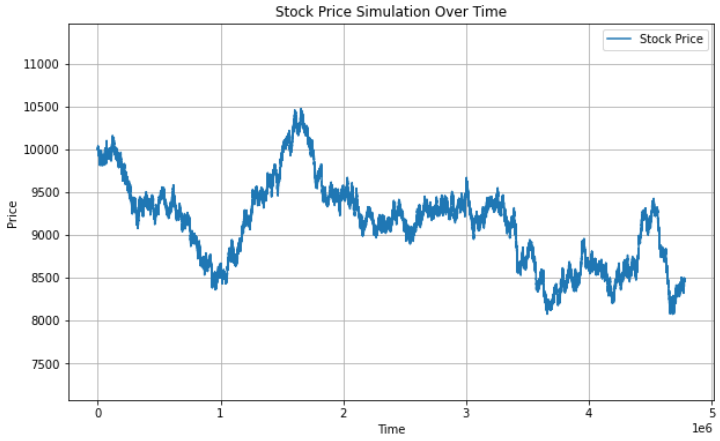


Figure 4: Run 2 of our baseline model.

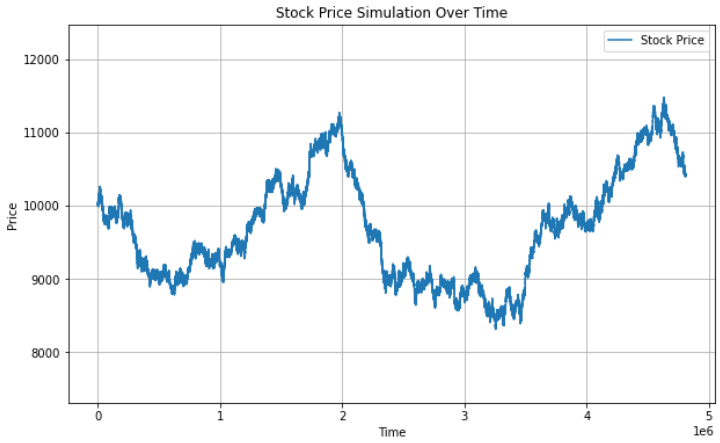


Figure 5: Run 3 of our baseline model.

As we can see from Figures 3-5, we have created an artificial model of a stock market. The normal agents have traded, based upon their predefined behaviours discussed in earlier sections, for 5000 time steps. There is a very interesting emergent behaviour across all runs of the model that we can observe from the stock prices. As we can see from the figures, initially in all models, there is an overall growth trend which begins to taper of in all models. In figure 3, we actually come to a steep decline in the price about halfway into the simulation, however this does stabilise back out eventually. The key thing to note about all three figures is that the stock price is rather stable. For all of the runs, it does not veer too far outside of 9000-11000 range for most of the simulation and all three end at a fairly similar price. To demonstrate this, we can see the average stock price of each simulation,

$$\text{Average Stock Price} = \begin{cases} 10670.8604, & \text{for Figure 3} \\ 9130.7457, & \text{for Figure 4} \\ 9693.6682, & \text{for Figure 5.} \end{cases}$$

This shows that the average stock price for our baseline simulation does not veer too far from the initial starting price of 10000, thus showing that a market consisting of just

normal agents is a stable market. So, what we can deduce from this is that, when left undisturbed by a more advanced class of investor, the NAs, which represent a more simple investor, have a fairly stable market which does fluctuate but does not veer too far from the mean price. This positions us nicely to investigate how a more advanced trader can influence the market.

3.5 Introducing The Learning Agent Class

We discussed earlier about how we are going to build a class of agent which are slightly more intelligent in the market than our class of NAs. These agents can learn from their previous trades, which is a very important ability in the world of trading. It is sometimes referred to as a 'trade review', which is when an investor reflects upon their previous trades, this allows them to hone their investing skills and perfect their approaches by refining and examining strategies. In our model, this is done by adjusting the weights of equation (4) and the learning process is explained by equation (7). We are now going to introduce this new class of agents into our baseline model to determine the effect of having a slightly more knowledgeable investor class in the market. To do this, we introduce a new class of agents in our code, known as ***'LearningAgent'***, and we define how they update the weights at each time step as a function within this class. The full code can be seen in Appendix Section B. In our model, we introduce 1000 NA2 agents. In real life there is likely to be fewer advanced general investors than there are regular investors, however for the sake of this model, we would like to demonstrate how an advanced investor can affect a stock price, thus we have introduced the same number of NA2s as we have NAs to give a more exaggerated outcome. One can of course adjust this and have differing numbers of NAs and NA2s. To adjust the number of NA2s, one simply increases the number of '$N\_learning$' in the function definition of ***'run_simulation'*** from the code in Appendix B. Here we are using the same parameters as the baseline, with the inclusion of $k = 4$. We again run our model for 3 simulations and we receive the following outputs:

Figure 6: Simulation 1 after introducing the learning agents.

Figure 7: Simulation 2 after introducing the learning agents.

Figure 8: Simulation 3 after introducing the learning agents.

As we can see from Figures 6-8, after running the simulation for 5000 time steps, the learning agent class has a profound effect on the stock price. Upon comparing to our baseline models from Section 3.4, where we only had NAs, Figures 3-5 showed very stable stock prices which over the course of the time steps did not deviate massively from the mean. On the other hand, Figures 6-8 show that by introducing the NA2s, the stock price actually has a negative trend overall. There are fluctuations in the price but over the course of the 5000 time steps, the stock price has a negative trend. To demonstrate this, we can again observe the average stock prices and compare them to that of our baseline models. Our average stock prices are:

$$\text{Average Stock Price} = \begin{cases} 6805.5545, & \text{for Figure 6} \\ 8143.2522, & \text{for Figure 7} \\ 8821.6583, & \text{for Figure 8.} \end{cases}$$

Therefore, we can see that the average stock price is lower for all simulations where we have included NA2s than all of the baseline models. In fact, the lowest average stock price that we have achieved from the baseline models is 9130.7457 which is considerably

greater than the highest stock price achieved by the NA2 simulations, which is 8821.6583. The average of our average stock prices for the baseline model is 9831.7581 which is just 168.2419 away from the starting price of 10000, which further demonstrates the stability of the NA market. The average of the average stock prices of the models where we have introduced NA2s is 7923.4883, which is 2076.5117 lower than our starting price of 10000, which shows how much more negative the market with a more advanced investor is. This may be because the advanced investors have more intelligent methods of determining their return, thus they may short sell more often to bank on a return in a negative market which thus reduces the share price.

3.6 The Technical Agents

One may now wonder how an incredibly advanced investor with a set investing ideology may affect the market. In the real life market, we often have hedge funds and big global banks which employ thousands of highly intelligent traders to conduct trades on the market. This has a profound effect on the market as they have a much larger amount of capital and a much wider knowledge of the market than a regular investor. To model this, we introduce a class of technical agents which rely upon more advanced technical analysis approaches to dictate their investment strategies. We will introduce a TA-m agent which is a momentum technical agent and a TA-r which is a reversal technical agent.

The way we will model this in our simulation is as follows, the n NA and NA2 agents will place their orders first and once the final one has placed their order, the TAs place their orders. We begin by explaining the behaviour of the TA-m agent. This is a momentum agent which, as the name suggests, follows the momentum of the stock price. This means that the agent buys the stock when the price is on the rise and sells the stock when it is on the decline. This means that the agent buys A shares at time t if $P^t > P^{t-t_m}$, and the agent will short sell A shares if $P^t < P^{t-t_m}$, where A is a pre-specified value which can be adjusted to represent the amount of resources the agent has, a larger value of A would mean they can buy more shares which represents an institution which has a larger share of the market. t_m is a value which is also pre-specified before running the simulation and it represents how far back the TA-m will look at the stock price in order to determine their investment strategies. Now this is similar to the investment strategy of the NAs as they can also look back in time to determine their expected return, the key difference here is that the TAs can adjust how far back in time they will look based upon learning from their previous trades. This value is t_m for the TA-m agent and t_r for the TA-r agent. Thus the TA-r agent is a reversal agent which trades against the momentum of the stock. This may represent various different types of institutions in real life such as a Hedge Fund, which often use short selling tactics to maximise returns against the momentum of the market. The TA-r buys A shares if $P^t < P^{t-t_r}$ and short sells A shares if $P^t > P^{t-t_r}$.

3.6.1 The Learning of The Technical Agents

The technical agents are similar to the NA2 agents in the sense that they have the ability to learn from previous trades. However, their approach is purely based on technical

analysis rather than the adjusting of weights between technical and fundamental analysis. Their learning approach is also more advanced to represent the more advanced investors they are modelled on. In the paper (Mizuta et al., 2022) the TAs conduct their learning through back-testing using Particle Swarm Optimisation (PSO), however we will take a slighty different approach for mathematical modelling simplicity. The TAs will adjust the values of t_r and t_m at each time step between the values of t_{min} and t_{max}, which will be specified before the model.

Over time, the agent will optimise their approach to arrive at the best possible values of t_r and t_m to maximise their return. In our mdoel, the TAs optimise these values using their specific historical performance. We will explain the learning process step by step, but first let us define:

$$t_x := \begin{cases} t_m, \text{ for TA-m} \\ t_r, \text{ for TA-r.} \end{cases}$$

So, initially, each TA will randomly select a value for t_m and t_r, respectively, from the interval $[t_{min}, t_{max}]$, and they will use this value of t_x for the next q time periods, this value is known as $t_{optimal}$. We will call q the *evaluation period* of the agent. Over the q time steps, the agents will keep a log where they are monitoring how each t_x value is performing. After q time steps, the TAs will perform an evaluation to see which value of t_x would have maximised their cumulative profit over the last q time steps, based upon the log they have kept. If the TAs have identified that a different value of t_x would have had a better return than their current value of $t_{optimal}$, then this better value of t_x becomes the new $t_{optimal}$. This continues iteratively for the duration of the simulation. For an example, say TA-m begins with $t_m = 5$ and it uses this value for the next $q = 100$ time steps. At t=100, it performs an evaluation and finds that $t_m = 15$ would have had a better return, it then uses this value for the next $q = 100$ time steps and so on. Mathematically, we can describe our learning process using a function:

$$f(t) : \ t \in [t_{min}, t_{max}] \ \longrightarrow \ \mathbb{R},$$

where $f(t)$ is the cumulative profit or loss associated with using a specific lag value t, for all t in our interval. Then we have that the optimisation process is:

$$t_{optimal} = \underset{t \ \in \ [t_{min}, t_{max}]}{\text{argmax}} \ f(t).$$

This means that the optimisation process is essentially just finding the value of t_x in the interval that would maximise the value of $f(t)$.

3.7 Introducing TAs to The Model

We will now introduce the TAs into our model from Section 3.5. There is first one important change we must make to our model that will allow us to properly investigate the impact of the TAs. Up until now, our models used a simple form of investigating the market impact of trades. Our baseline model and our model from Section 3.5, adjusted the stock price up or down by 1 regardless of how many shares were purchased or sold, respectively. This was not an issue in those models as both NAs and NA2s could only

purchase 1 share at each time step. Our TAs can purchase A shares at each time step as they are used to model financial institutions with a larger amount of capital to invest in the market. Thus we must adjust how our stock price is updated.

To do this, we make modifications to our code, which can be seen in full in the appendices. Firstly, we adjust the way we have defined ***'execute_order'*** in our code to take three arguments, rather than two, this new argument is *'shares'* which defaults to 1 unless stated otherwise. This argument represents the number of shares purchased by the agent. We then set our TAs to purchase *'shares = self.A'* shares, which ensures that they purchase A shares whilst the other agents continue to purchase 1 share. We then modify ***'execute_order'*** so that our stock price is adjusted up or down at each time step, depending upon the number of stocks purchased or sold respectively at time t. To increase realism in the market, we also make it so that the stock price is updated by $b \times$ shares in our formula for adjusting share prices. To ensure that prices are not so volatile one would choose a small value of b less than or equal to 0.5, but in our model we use 1 to exacerbate the effect of individual trades, so that we can analyse the effects of the inclusion of our agents more clearly.

We will now add in our TAs, beginning with TA-m and then TA-r, and then both to investigate the effect of each agent. To do this, we set $t_{min} = 5$, $t_{max} = 100$, $q = 100$ and $A = 100$.

3.7.1 Adding in TA-m

We now include our momentum agent TA-m. This agent will buy the asset when the price is rising and sell when the price is declining. The TA-m agent is made to represent more traditional financial institutions, such as large investment banks and asset management firms, which will often trade in a less risky manner, so the momentum strategy is an appropriate model for them. We again run our simulation three times and receive the following outputs:

Figure 9: Simulation 1 after introducing TA-m.

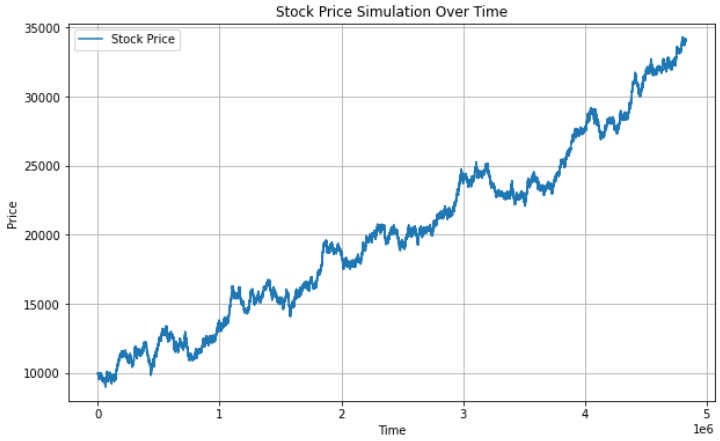


Figure 10: Simulation 2 after introducing TA-m.

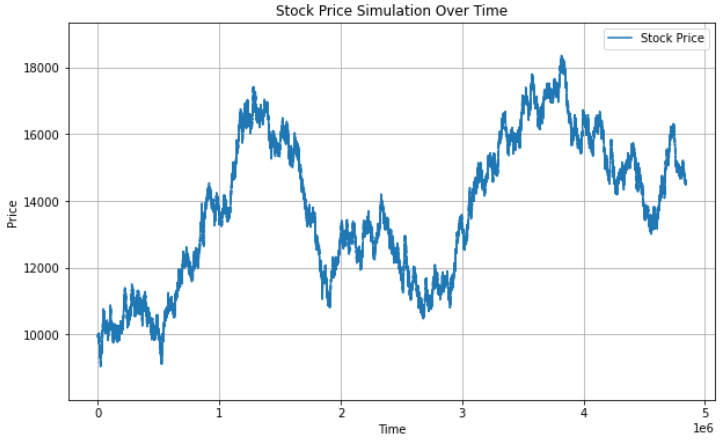


Figure 11: Simulation 3 after introducing TA-m.

As we can see from Figures 9-11, introducing the more advanced momentum agent has a profound effect on the stock price. In all off our figures, we see that the inclusion of TA-m causes the price of the asset to rise massively over time. There is some variation as to the early effects of introducing the agent, i.e. what occurs before t=1000, but this can variation is due to the random nature of the initial value of t_m. Despite the initial variation, the overall trend is clear, and identical in all 3 simulations, TA-m's momentum strategy causes a growth in stock price over time despite fluctuations. This can be seen from the average prices of each simulation which are:

$$\text{Average Stock Price} = \begin{cases} 12565.1872, & \text{for Figure 9} \\ 20080.4766, & \text{for Figure 10} \\ 13793.3311, & \text{for Figure 11.} \end{cases}$$

This shows that the average stock price with the inclusion of TA-m is considerably higher than the initial stock price of 10000, and is much higher than both our baseline and our learning agent model. Particularly in Figure 10, we can see that our stock price has broken the 20000 mark for the first time. This shows that the TA-m agent causes a price hike to occur when they are in the market.

3.7.2 Adding in TA-r

We will TA-r agent, but we will exclude the TA-m agent, as we are primarily interested in the effect of TA-r on the price, in this sub-section. This agent employs a reversal investment strategy, which is counter to the market trend. Thus, they buy when the price rises and sells when the price falls. This agent has a riskier investment strategy and is meant to represent financial institutions with a higher tolerance for risky (and perhaps unconventional) investment strategies, such as hedge funds and boutique trading firms. We again run our simulation three times and receive the following outputs:

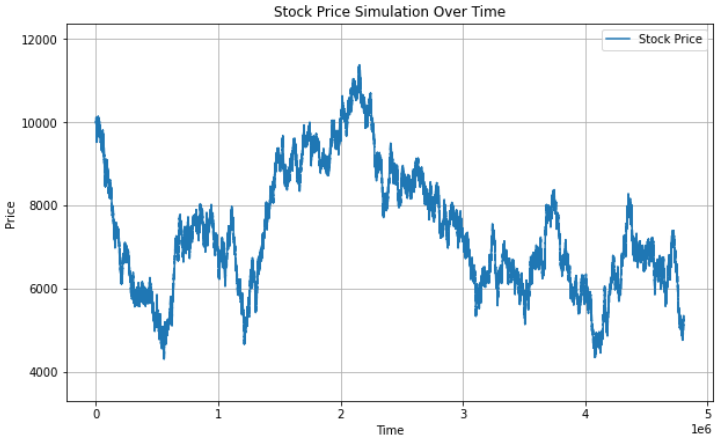


Figure 12: Simulation 1 after introducing TA-r.

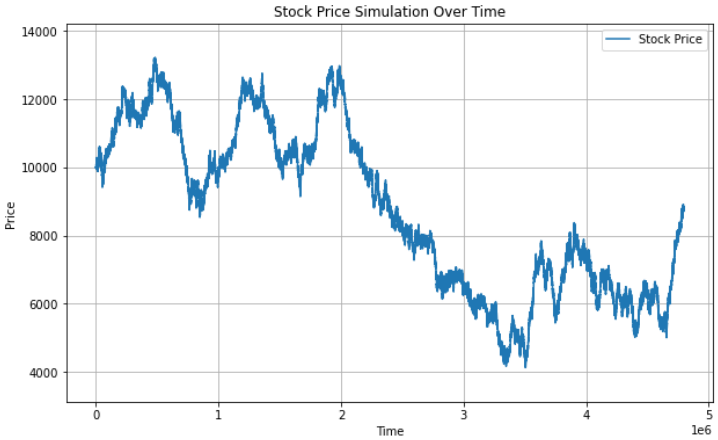


Figure 13: Simulation 2 after introducing TA-r.

Figure 14: Simulation 3 after introducing TA-r.

As we can see from Figures 12-14, the inclusion of TA-r is the exact contrast of the inclusion of TA-m. This is because TA-r has the exact opposite effect on the market. Whilst TA-m caused a hike in the stock price, TA-r has caused a massive decline in the stock price in all three simulations. This is in line with what is to be expected as aggressive counter-market short selling often has a negative effect on stock prices. Similar to the inclusion of TA-m, there is variation in the initial ($t < 1000$) stock price, in Figure 12 it declines massively initially, whilst it increases slightly in both Figure 13 and 14. This is again caused by variations in the initial value of t_r due to the random nature of the initial part of the learning process. Despite the initial variations, and the fluctuations in price throughout, the initial trend is clear, TA-r's reversal strategy causes the asset price to reduce substantially over time. This can be seen from the average stock prices for each simulation, which are:

$$\text{Average Stock Price} = \begin{cases} 7357.0237, & \text{for Figure 9} \\ 8743.2509, & \text{for Figure 10} \\ 6812.3559, & \text{for Figure 11.} \end{cases}$$

This shows that the inclusion of TA-r causes the average stock price to be significantly lower than the initial price of 10000, it is also significantly lower than our baseline model. The negative nature of the model is somewhat similar to the effect of the NA2 agents in Section 3.5, albeit at a much higher rate of decline in the asset price, possibly due to the higher number of shares sold/purchased at each time step. This negative effect is most apparent in Figure 14, where our stock dropped below 2000 for the first time in any simulation. This shows that overall our TA-r has a negative effect on the stock price, and causes a decline in the price to occur when they are included in the market.

3.7.3 Including Both TA-r And TA-m

As TA-r and TA-m have opposing effects on the market, a natural question would be: what effect would we have if we were to include both agents? We would imagine that they would potentially cancel out, leading to a very stable asset price. Thus, we run our simulation 3 times, now with the inclusion of both a TA-r agent and a TA-m agent, to receive the following outputs:

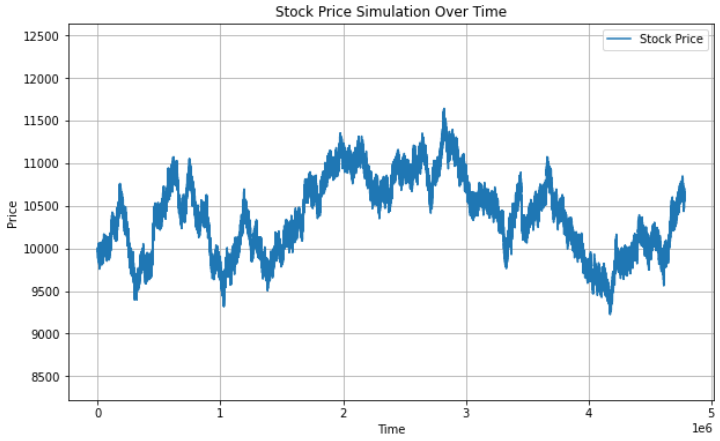


Figure 15: Simulation 1 after introducing TA-r and TA-m.

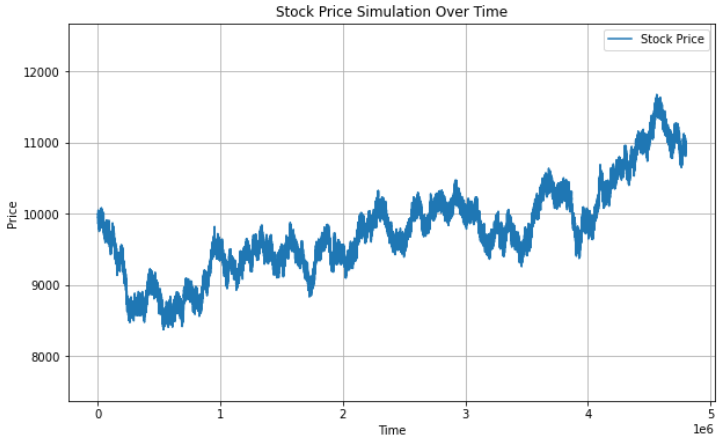


Figure 16: Simulation 2 after introducing TA-r and TA-m.

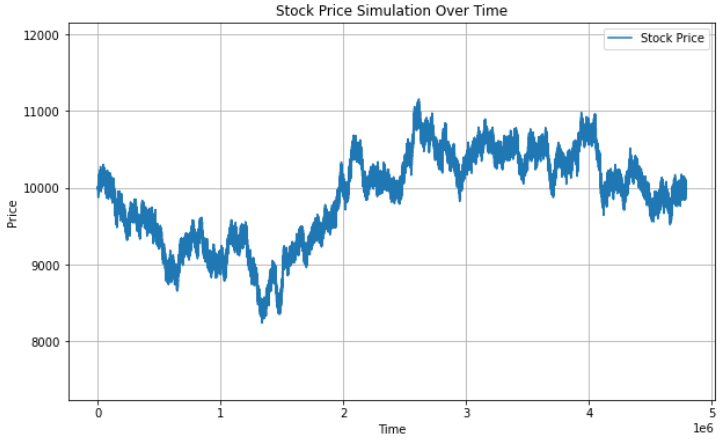


Figure 17: Simulation 3 after introducing TA-r and TA-m.

As we can see from Figures 15-17, we indeed do have a very stable market with the inclusion of all of our agents. There are fluctuations in the price of the asset, but the vast

majority of these fluctuations are between 9000 and 11000, thus the stock price never strays too far from the initial value of 10000. This can be seen from the average stock prices of our simulations, which are:

$$\text{Average Stock Price} = \begin{cases} 10378.8518, \text{ for Figure 15} \\ 9757.8316, \text{ for Figure 16} \\ 9884.2922, \text{ for Figure 17.} \end{cases}$$

This shows that the average stock price is incredibly close to the initial value of 10000, in fact the average variation from 10000 in our simulation is only 245.576. This means that on average across our simulations, the average stock price varies from the initial price by only 245.576.

3.8 Brief Explanation of Our Code

In this section, we will briefly touch on what is happening in our code, which can be viewed in the appendix. Initially we define our class of agents, we have four classes: 'NormalAgent', 'LearningAgent', 'TechnicalMomentumAgent' and 'TechnicalReversalAgent'. In each class, we define their behaviours, what motivates them to buy or sell and (for our final model only) we state the number of shares they trade at each time step. For the 'LearningAgent' class, their behaviour is identical to the 'NormalAgent' class, but we include a definition of how they adjust the weights at each time step. Similarly, for both Technical Agent classes, we define how they adjust t_x at each time step. We then build our environment for our agents to operate in, this is the 'StockMarketModel' section of our code. Within this, we define the values of our constants, set which agents can operate in the market and most importantly, we define how the price updates at each time step within the 'execute_order' part of this section of code. Finally, we run the simulation by choosing the number of time steps and each type of agent, and our we create a graph of the stock price as our output.

3.9 Conclusion of Market Model

The model including all four agent classes is the most stable simulation. This can be seen in the following plot:

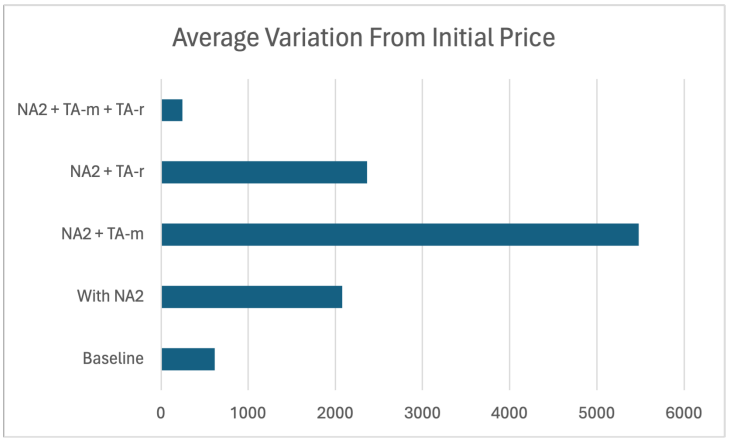


Figure 18: Plot of the average absolute value of the variance from the initial price for each model (note all models also include NAs).

From Figure 18, we can see the average variance from the initial value of 10000 for each model, this allows us to see which model has the most stable market. The values in the chart were calculated by taking

$$\frac{|(10000 - \text{average stock price for each figure})|}{3}$$

for each model type. We can see that the model which includes all four agent types (NA, NA2, TA-m and TA-r) has by far the smallest variation from the initial price on average, showing that it is indeed the most stable market. So we can conclude from our model, the market is somewhat stable with just the inclusion of NAs, by adding in the NA2 class we have an unstable market which varies massively from 1000 often in the negative direction, by including a single TA-m the price again varies massively from 10000 to the positive direction (in fact this model has the largest variation by far), by including a TA-r and no TA-m we again have instability and large variation but in the negative direction, and finally by including all agents we have the most stable market with a very small variation.

3.10 Discussion of The Models

We have been able to demonstrate how ABM can be used to model a financial market, and thus we have been able to reinforce its importance as a forecasting tool for economists and traders. Our model, however, may not have much use case in the world of trading, as it stands. Thus, this section serves as a discussion about how our model can be built upon to be even more realistic, which could allow traders to properly estimate the effects of their trades.

Firstly, in our model, we have four types of agents each with their own approaches in the market. To make this model more realistic, we could include even more agents which can represent different types of investors. We have generalised the NAs to represent a layman investor, the NA2s to be a slightly smarter investor, TA-m to represent traditional investment banks and institutions similar to them, and the TA-r to represent hedge

funds and traders with a higher tolerance to risk. One could also model other investors which may be important for determining market dynamics, some examples may include investment banks of different sizes (bulge bracket vs boutique vs middle market) or long term fundamental investors. There are many types of investors in real life, and the effects of them on the market are important for studying how an asset price moves.

Secondly, we could improve the learning process of the TAs. In our model, we took a fairly simple learning process for mathematical simplicity, however one could implement a more advanced learning alogrithm to create an even more intelligent and realistic investor who learns from every trade. An example framework could be based upon particle swarm optimisation which is used in (Mizuta et al., 2022), this would allow the agents to gradually improve and narrow down the best values of t_x. Another idea could be building an even more intelligent, cutting edge agent using AI. We discussed in Section 1.2 about how AI is revolutionising ABM, and its implementation here could allow for a hyper-realistic market model. If a savvy coder was to create an agent that was capable of using AI to learn from its previous trades and to anticipate the actions of other traders, it would be incredibly interesting to study the effect this agent has on our model.

Thirdly, we have run each of our simulations three times. To improve this, one could run the model thousands or millions of times and then plot the average value of the asset at each time step onto a line graph to provide the most accurate estimation of the market behaviour. This would require a great deal of computing power and time but it would help to eliminate any outliers and build a much more reliable and accurate estimation of the market over time.

Finally, our model purely considers market impact to determine future asset prices, but as any decent investor knows, the price of a stock is influenced by many external factors such as industry news and trends, inflation, the global forex market, the price of oil and many, many other factors. To build upon our model, we could implement more randomness to account for industry news and world events, we could include the effect of inflation by reducing the value of our currency continuously over time and we can model many important factors to increase the realism of our model. One may even want to model our ABM as a stochastic finite dynamical system, as we have discussed earlier. The model we have created simply serves as a framework to show ABM in practice, and we have been able to demonstrate that the theory does indeed work. There are many ways that our model can be improved, which we have detailed, to allow us to build a hyper-realistic ABM of an asset price.

4 Aside: Social Network Analysis

In this section, we introduce another popular approach used in Computational Social Science. We will not go into nearly as much detail as we have on ABM, as this paper is primarily only focused on ABM, this section merely serves as a guideline to demonstrate how advanced Computational Social Science is by showing the breadth of approaches available. Networks are usually modelled as graphs, where we have a set of vertices

$V = (v_1, v_2, ..., v_n)$ which represent individuals or groups (called actors in the system), such as financial institutions, and a set of edges $E = (e_1, e_2, ..., e_m)$ which represent the ties between the vertices. These ties can represent interactions between actors, or similarities between them or whatever the relation between the agents that is being modelled. The main goal of Social Network Analysis (SNA) is to examine both the contents and patterns of the relationships between actors, rather than the social entities themselves, in order to understand the relations between actors and their implications (Tabassum et al., 2018). We could have a single set of nodes E which have distinct relations between them, or we could have distinct relations between multiple sets of nodes $(E_1, E_2, ..., E_p)$. When $p \geq 2$, i.e. we have more than one set of nodes, the network is known as a heterogeneous network. An important heterogeneous network to note is when $p = 2$, this is known as a two-mode network. Two-mode networks are important as they can represent a great number of social systems, for example user-product networks such as Amazon, or actor-movies networks such as IMDB (Tabassum et al., 2018).

SNA implements methods from Graph Theory and Network Analysis in order to make conclusions about social structures. It has various applications in finance, for example it can be used in risk analysis as through analysing the structure and relationships of a network, one can identify clusters of organisations or investors who are linked, therefore being able to understand how failure or financial distress of one actor can affect many other key players in the network. It can also be used to diversify an investment portfolio as SNA can identify which investments are linked to each other, or conversly which are independent of each other allowing for a more diverse, and thus risk-free, portfolio. It is another interesting modelling approach, with many similarities to ABM.

5 Conclusion

The goal of this paper was to detail ABM and show that it is indeed a viable and emerging alternative to traditional modelling approaches. We have introduced ABM, explained its origins from Cellular Automata to discussions of the future of AI based ABM, we have shown simple examples, using Python and Mesa, of Conway's Game of Life and we have built our own model to show the effectiveness of ABM in finance and economics. Agent-Based Modelling has a lot of room for growth in the future, as we develop more and more advanced computers, and it has endless use-cases. In this report, we narrowed down to only one, that being finance, however the possibilities of ABM are endless and there are many impressive models out there where researchers have used ABM to model many different things, including traffic patterns, biology, human behaviour, war, disease and many other fields. To conclude, Agent-Based Modelling is a promising and rapidly emerging modelling approach with applications in many different fields, and as technology develops, one can be certain that ABMs emergence into the mainstream of modelling approaches is but an inevitability.

References

Axtell, Robert L and J Doyne Farmer (2022), "Agent-based modeling in economics and finance: Past, present, and future." *Journal of Economic Literature*.

Bianchi, Federico and Flaminio Squazzoni (2015), "Agent-based models in sociology." *Wiley Interdisciplinary Reviews: Computational Statistics*, 7, 284–306.

Conte, Rosaria, Nigel Gilbert, Giulia Bonelli, Claudio Cioffi-Revilla, Guillaume Deffuant, Janos Kertesz, Vittorio Loreto, Suzy Moat, J P Nadal, Anxo Sanchez, et al. (2012), "Manifesto of computational social science." *The European Physical Journal Special Topics*, 214, 325–346.

Hamill, Lynne and Nigel Gilbert (2015), *Agent-based modelling in economics*. John Wiley & Sons.

Hanappi, Hardy (2017), "Agent-based modelling. history, essence, future." *PSL Quarterly Review*, 70, 449–472.

Heath, Brian L and Ray R Hill (2008), "The early history of agent-based modeling." In *IIE Annual Conference. Proceedings*, 971, Institute of Industrial and Systems Engineers (IISE).

Kazil, Jackie, David Masad, and Andrew Crooks (2020), "Utilizing python for agent-based modeling: The mesa framework." In *Social, Cultural, and Behavioral Modeling* (Robert Thomson, Halil Bisgin, Christopher Dancy, Ayaz Hyder, and Muhammad Hussain, eds.), 308–317, Springer International Publishing, Cham.

Laubenbacher, Reinhard, Abdul S Jarrah, Henning Mortveit, and SS Ravi (2007), "A mathematical formalism for agent-based modeling." *arXiv preprint arXiv:0801.0249*.

Mizuta, Takanobu, Kiyoshi Izumi, and Shinobu Yoshimura (2013), "Price variation limits and financial market bubbles: Artificial market simulations with agents' learning process." In *2013 IEEE Conference on Computational Intelligence for Financial Engineering & Economics (CIFEr)*, 1–7, IEEE.

Mizuta, Takanobu, Isao Yagi, and Kosei Takashima (2022), "Instability of financial markets by optimizing investment strategies investigated by an agent-based model." In *2022 IEEE Symposium on Computational Intelligence for Financial Engineering and Economics (CIFEr)*, 1–8, IEEE.

Tabassum, Shazia, Fabiola SF Pereira, Sofia Fernandes, and João Gama (2018), "Social network analysis: An overview." *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8, e1256.

Tarver, Evan (2021), "Auction market: Definition, how it works in trading, and examples." Website, URL `https://www.investopedia.com/terms/a/auctionmarket.asp`. Last checked: 05/03/2024.

Vayadande, Kuldeep, Ritesh Pokarne, Mahalakshmi Phaldesai, Tanushri Bhuruk, Tanmay Patil, and Prachi Kumar (2022), "Simulation of conway's game of life using cellular automata." *SIMULATION*, 9.

Yagi, Isao, Mahiro Hoshino, Takanobu Mizuta, et al. (2023), "Impact of high-frequency trading with an order book imbalance strategy on agent-based stock markets." *Complexity*, 2023.

Appendices

A Code of Baseline Model

We now include the code for the baseline model from Section 3.4:

```
import numpy as np
import matplotlib.pyplot as plt
from mesa import Model, Agent
from mesa.time import BaseScheduler


class NormalAgent(Agent):
    def __init__(self, unique_id, model):
        super().__init__(unique_id, model)
        self.w1 = np.random.uniform(0, 1)
        self.w2 = np.random.uniform(0, 100)
        self.w3 = np.random.uniform(0, 1)
        self.tau = np.random.randint(1, model.tau_max + 1)
        self.epsilon = np.random.normal(0, model.sigma_epsilon)

    def step(self):
        if len(self.model.P) > self.tau:
            P_t_minus_tau_minus_1 = self.model.P[-self.tau - 1]
        else:
            P_t_minus_tau_minus_1 = self.model.P[0]
        Pf = self.model.Pf
        P_t_minus_1 = self.model.P[-1]
        numerator = (
            self.w1 * np.log(Pf / P_t_minus_1) +
            self.w2 * np.log(P_t_minus_1 / P_t_minus_tau_minus_1) +
            self.w3 * self.epsilon
        )
        denominator = self.w1 + self.w2 + self.w3
        r_e_j_t = numerator / denominator
        P_e_j_t = P_t_minus_1 * np.exp(r_e_j_t)
        rho_j_t = np.random.rand()
        P_o_j_t = P_e_j_t + self.model.Pd * (2 * rho_j_t - 1)
        if P_e_j_t > P_o_j_t:
            self.model.execute_order('buy')
        elif P_e_j_t < P_o_j_t:
            self.model.execute_order('sell')

```

```
38
39 class StockMarketModel(Model):
40     def __init__(self, N=1000):
41         self.num_agents = N
42         self.schedule = BaseScheduler(self)
43         self.Pf = 10000
44         self.Pd = 1000
45         self.tau_max = 10000
46         self.sigma_epsilon = 0.03
47         self.P = [10000]
48         for i in range(self.num_agents):
49             a = NormalAgent(i, self)
50             self.schedule.add(a)
51
52     def execute_order(self, order_type):
53         if order_type == 'buy':
54             self.P.append(self.P[-1] + 1)
55         elif order_type == 'sell':
56             self.P.append(self.P[-1] - 1)
57
58     def step(self):
59         for agent in sorted(self.schedule.agents, key=lambda a: a.unique_id
    ):
60             agent.step()
61         # Print the stock price at the current step
62         print(f'Stock Price at step {len(self.P) - 1}: {self.P[-1]}')
63
64
65 def run_simulation(steps=5000):
66     model = StockMarketModel()
67     for i in range(steps):
68         model.step()
69
70     plt.figure(figsize=(10, 6))
71     plt.plot(model.P, label='Stock Price')
72     plt.title('Stock Price Simulation Over Time')
73     plt.xlabel('Time')
74     plt.ylabel('Price')
75     plt.legend()
76     plt.grid(True)
77     plt.ylim([min(model.P) - 1000, max(model.P) + 1000])
78     plt.show()
79     average_stock_price = np.mean(model.P)
80     print(f'Average Stock Price: {average_stock_price}')
81
82 run_simulation(steps=5000)
```

B Code For The Inclusion Of The Learning Agents

We now include the code for the model where we have included the NA2 agents in Section 3.5:

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

```python
from mesa import Model, Agent
from mesa.time import BaseScheduler


class NormalAgent(Agent):
    def __init__(self, unique_id, model):
        super().__init__(unique_id, model)
        self.w1 = np.random.uniform(0, 1)
        self.w2 = np.random.uniform(0, 100)
        self.w3 = np.random.uniform(0, 1)
        self.tau = np.random.randint(1, model.tau_max + 1)
        self.epsilon = np.random.normal(0, model.sigma_epsilon)

    def step(self):
        if len(self.model.P) > self.tau:
            P_t_minus_tau_minus_1 = self.model.P[-self.tau - 1]
        else:
            P_t_minus_tau_minus_1 = self.model.P[0]
        Pf = self.model.Pf
        P_t_minus_1 = self.model.P[-1]
        numerator = (
            self.w1 * np.log(Pf / P_t_minus_1) +
            self.w2 * np.log(P_t_minus_1 / P_t_minus_tau_minus_1) +
            self.w3 * self.epsilon
        )
        denominator = self.w1 + self.w2 + self.w3
        r_e_j_t = numerator / denominator
        P_e_j_t = P_t_minus_1 * np.exp(r_e_j_t)
        rho_j_t = np.random.rand()
        P_o_j_t = P_e_j_t + self.model.Pd * (2 * rho_j_t - 1)
        if P_e_j_t > P_o_j_t:
            self.model.execute_order('buy')
        elif P_e_j_t < P_o_j_t:
            self.model.execute_order('sell')


class LearningAgent(NormalAgent):
    def __init__(self, unique_id, model, k=4):
        super().__init__(unique_id, model)
        self.k = k

    def adjust_weights(self, r_l_t):
        u_j_t = np.random.uniform(0, 1)
        weights = [self.w1, self.w2, self.w3]
        max_weights = [1, 100, 1]  # w1_max, w2_max, w3_max
        for i in range(3):
            if np.sign(r_l_t) == np.sign(self.epsilon):
                weights[i] += self.k * r_l_t * u_j_t * \
                    (max_weights[i] - weights[i])
            else:
                weights[i] -= self.k * r_l_t * u_j_t * weights[i]
            # Ensure weights remain within bounds
            weights[i] = max(min(weights[i], max_weights[i]), 0)
        self.w1, self.w2, self.w3 = weights

```

```
    def step(self):
        super().step()
        if len(self.model.P) > 100:
            P_t = self.model.P[-1]
            P_t_t_l = self.model.P[-101]
            r_l_t = np.log(P_t / P_t_t_l)
            self.adjust_weights(r_l_t)


class StockMarketModel(Model):
    def __init__(self, N=1000, N_learning=1000):
        self.num_agents = N
        self.schedule = BaseScheduler(self)
        self.Pf = 10000
        self.Pd = 1000
        self.tau_max = 10000
        self.sigma_epsilon = 0.03
        self.P = [10000]
        for i in range(self.num_agents - N_learning):
            a = NormalAgent(i, self)
            self.schedule.add(a)
        for i in range(self.num_agents - N_learning, self.num_agents):
            a = LearningAgent(i, self)
            self.schedule.add(a)

    def execute_order(self, order_type):
        if order_type == 'buy':
            self.P.append(self.P[-1] + 1)
        elif order_type == 'sell':
            self.P.append(self.P[-1] - 1)

    def step(self):
        for agent in sorted(self.schedule.agents, key=lambda a: a.unique_id
    ):
            agent.step()
        print(f'Stock Price at step {len(self.P) - 1}: {self.P[-1]}')


def run_simulation(steps=5000):
    model = StockMarketModel(N=1000, N_learning=100)
    for i in range(steps):
        model.step()

    plt.figure(figsize=(10, 6))
    plt.plot(model.P, label='Stock Price')
    plt.title('Stock Price Simulation Over Time')
    plt.xlabel('Time')
    plt.ylabel('Price')
    plt.legend()
    plt.grid(True)
    # Adjust y-axis scale as needed
    plt.ylim([min(model.P) - 1000, max(model.P) + 1000])
    plt.show()
    average_stock_price = np.mean(model.P)
    print(f'Average Stock Price: {average_stock_price}')
```

```
112
113
114 run_simulation(steps=5000)
```

C Code For The Inclusion of The Technical Agents

This is our code for including the technical agents. The code in this section shows the script for the inclusion of both TA-m and TA-r. However, if one would like to exclude one or the other as we have done in Sections 3.7.1 and 3.7.2, one would adjust 'N_technical' or 'N_reversal' to 0 to exclude TA-m or TA-r, respectively. Here is the full code:

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from mesa import Model, Agent
4  from mesa.time import BaseScheduler
5
6
7  class NormalAgent(Agent):
8      def __init__(self, unique_id, model):
9          super().__init__(unique_id, model)
10         self.w1 = np.random.uniform(0, 1)
11         self.w2 = np.random.uniform(0, 100)
12         self.w3 = np.random.uniform(0, 1)
13         self.tau = np.random.randint(1, model.tau_max + 1)
14         self.epsilon = np.random.normal(0, model.sigma_epsilon)
15
16     def step(self):
17         if len(self.model.P) > self.tau:
18             P_t_minus_tau_minus_1 = self.model.P[-self.tau - 1]
19         else:
20             P_t_minus_tau_minus_1 = self.model.P[0]
21         Pf = self.model.Pf
22         P_t_minus_1 = self.model.P[-1]
23         numerator = (
24             self.w1 * np.log(Pf / P_t_minus_1) +
25             self.w2 * np.log(P_t_minus_1 / P_t_minus_tau_minus_1) +
26             self.w3 * self.epsilon
27         )
28         denominator = self.w1 + self.w2 + self.w3
29         r_e_j_t = numerator / denominator
30         P_e_j_t = P_t_minus_1 * np.exp(r_e_j_t)
31         rho_j_t = np.random.rand()
32         P_o_j_t = P_e_j_t + self.model.Pd * (2 * rho_j_t - 1)
33         if P_e_j_t > P_o_j_t:
34             self.model.execute_order('buy', 1)
35         elif P_e_j_t < P_o_j_t:
36             self.model.execute_order('sell', 1)
37
38
39 class LearningAgent(NormalAgent):
40     def __init__(self, unique_id, model, k=4):
41         super().__init__(unique_id, model)
42         self.k = k
43
```

```
    def adjust_weights(self, r_l_t):
        u_j_t = np.random.uniform(0, 1)
        weights = [self.w1, self.w2, self.w3]
        max_weights = [1, 100, 1]  # w1_max, w2_max, w3_max
        for i in range(3):
            if np.sign(r_l_t) == np.sign(self.epsilon):
                weights[i] += self.k * r_l_t * u_j_t * \
                    (max_weights[i] - weights[i])
            else:
                weights[i] -= self.k * r_l_t * u_j_t * weights[i]
            # Ensure weights remain within bounds
            weights[i] = max(min(weights[i], max_weights[i]), 0)
        self.w1, self.w2, self.w3 = weights

    def step(self):
        super().step()
        if len(self.model.P) > 100:
            P_t = self.model.P[-1]
            P_t_t_l = self.model.P[-101]
            r_l_t = np.log(P_t / P_t_t_l)
            self.adjust_weights(r_l_t)

class TechnicalMomentumAgent(Agent):
    def __init__(self, unique_id, model, A=100, evaluation_period=100):
        super().__init__(unique_id, model)
        self.A = A
        self.evaluation_period = evaluation_period
        self.tm_range = np.arange(5, 101)
        self.best_tm = np.random.choice(self.tm_range)
        self.performance_record = {tm: 0 for tm in self.tm_range}

    def evaluate_tm(self):
        best_performance = -np.inf
        for tm, performance in self.performance_record.items():
            if performance > best_performance:
                best_performance = performance
                self.best_tm = tm
        # Reset performance record for the next evaluation period
        self.performance_record = {tm: 0 for tm in self.tm_range}

    def step(self):
        if len(self.model.P) > self.best_tm:
            P_t = self.model.P[-1]
            P_t_tm = self.model.P[-self.best_tm - 1]
            if P_t > P_t_tm:
                self.model.execute_order('buy', self.A)
                self.performance_record[self.best_tm] += (P_t - P_t_tm)
            elif P_t < P_t_tm:
                self.model.execute_order('sell', self.A)
                self.performance_record[self.best_tm] += (P_t_tm - P_t)

            # Evaluate and adjust t_m every evaluation_period steps
            if len(self.model.P) % self.evaluation_period == 0:
                self.evaluate_tm()

```

```python
class TechnicalReversalAgent(Agent):
    def __init__(self, unique_id, model, A=100, evaluation_period=100):
        super().__init__(unique_id, model)
        self.A = A
        self.evaluation_period = evaluation_period
        self.tr_range = np.arange(5, 101)
        self.best_tr = np.random.choice(self.tr_range)
        self.performance_record = {tr: 0 for tr in self.tr_range}

    def evaluate_tr(self):
        best_performance = -np.inf
        for tr, performance in self.performance_record.items():
            if performance > best_performance:
                best_performance = performance
                self.best_tr = tr
        # Reset performance record for the next evaluation period
        self.performance_record = {tr: 0 for tr in self.tr_range}

    def step(self):
        if len(self.model.P) > self.best_tr:
            P_t = self.model.P[-1]
            P_t_tr = self.model.P[-self.best_tr - 1]
            if P_t < P_t_tr:
                self.model.execute_order('buy', self.A)
                self.performance_record[self.best_tr] += (P_t_tr - P_t)
            elif P_t > P_t_tr:
                self.model.execute_order('sell', self.A)
                self.performance_record[self.best_tr] += (P_t - P_t_tr)

            # Evaluate and adjust t_r every evaluation_period steps
            if len(self.model.P) % self.evaluation_period == 0:
                self.evaluate_tr()


class StockMarketModel(Model):
    def __init__(self, N=1000, N_learning=1000, N_technical=1, N_reversal
    =1):
        self.num_agents = N
        self.schedule = BaseScheduler(self)
        self.Pf = 10000
        self.Pd = 1000
        self.tau_max = 10000
        self.sigma_epsilon = 0.03
        self.P = [10000]
        for i in range(self.num_agents - N_learning):
            a = NormalAgent(i, self)
            self.schedule.add(a)
        for i in range(self.num_agents - N_learning, self.num_agents):
            a = LearningAgent(i, self)
            self.schedule.add(a)
        for _ in range(N_technical):
            a = TechnicalMomentumAgent(self.num_agents, self)
            self.schedule.add(a)
        for _ in range(N_reversal):
            a = TechnicalReversalAgent(self.num_agents + N_technical, self)
```

```
153             self.schedule.add(a)
154
155
156     def execute_order(self, order_type, shares=1):
157         if order_type == 'buy':
158             self.P.append(self.P[-1] + shares * 1)  # Adjusted for shares
159         elif order_type == 'sell':
160             self.P.append(self.P[-1] - shares * 1)
161
162     def step(self):
163         for agent in sorted(self.schedule.agents, key=lambda a: a.unique_id
    ):
164             agent.step()
165         print(f'Stock Price at step {len(self.P) - 1}: {self.P[-1]}')
166
167
168 def run_simulation(steps=5000):
169     model = StockMarketModel(N=1000, N_learning=100, N_technical=1,
170                              N_reversal=1)
171     for i in range(steps):
172         model.step()
173
174     plt.figure(figsize=(10, 6))
175     plt.plot(model.P, label='Stock Price')
176     plt.title('Stock Price Simulation Over Time')
177     plt.xlabel('Time')
178     plt.ylabel('Price')
179     plt.legend()
180     plt.grid(True)
181     # Adjust y-axis scale as needed
182     plt.ylim([min(model.P) - 1000, max(model.P) + 1000])
183     plt.show()
184     average_stock_price = np.mean(model.P)
185     print(f'Average Stock Price: {average_stock_price}')
186
187
188 run_simulation(steps=5000)
```