oTree Markets

Morgan Grant

March 16, 2020

Abstract

This paper presents oTree Markets, a flexible framework for the construction of market simulation experiments in oTree. oTree Markets provides three components: a Python implementation of a Continuous Double Auction exchange, a reference text-based interface and an oTree layer for communication and state management. These three components are designed with modularity in mind, with the intent that each of them could be replaced with modified versions to suit the needs of a wide variety of market simulation experiments.

1 Introduction

Market simulation is an essential tool used by experimental economists to study markets and evaluate their efficiency and effectiveness. It enables the experimental study of markets under controlled conditions and allows researchers to answer a broader variety of questions about markets than can be answered with statistical analysis of collected market data.

There are limited options for those who wish to create these types of market simulation experiments. Flex e-Markets is one such tool for the creation of market simulations, but it is closed-source software intended for use in industry and may not be suitable for academic market researchers. Additionally, it being closed-source means that its features are limited to those provided by the authors and that some types of simulations are not possible to create. Other simulation options are similarly limited.

Ideally, experimenters would be able to create market simulation experiments in a familiar programming environment. It is likely that most experimental economists are familiar with oTree as it is the current state of the art in economics experiment frameworks. oTree Markets is designed to solve these two problems: it is open source and flexible enough to encompass a wide variety of simulation experiments, and it allows experimenters to build their market experiments in oTree rather than in an unfamiliar software setting.

This project was created with the assistance of UC Santa Cruz's LEEPS Lab. LEEPS has been studying experimental economics for many years and has lacked an effective, extensible framework for the creation of market simulation experiments. Advice from LEEPS' experimenters was invaluable for guiding this project's design towards a framework of maximal usefulness.

2 Motivation

Market design is a critical area of focus for economics. Financial markets have a huge influence on the economic function of our society, and as such there is an urgent need to understand the mechanisms by which they operate. The field of market design analyzes markets, financial and otherwise, evaluating them along vectors such as fairness, efficiency and effectiveness.

This need for market analysis has become especially apparent with the advent of computerized communication and trading technologies. These new technologies have fundamentally changed the way in which markets operate. As a result, market designs which were previously thought to be effective do not operate in the same way under these new conditions and must be reconsidered. Additionally, computerization introduces possibilities for new market environments which were not previously possible

2.1 Market Simulation

How can we quantitatively investigate these questions about markets? One could take a descriptive

approach by analyzing recorded trading data and looking for correlations between sets of market parameters and outcomes. This can be an effective technique, but it is limited in that it can only analyze market designs which already exist and are being used in real trading environments. Additionally many market institutions only provide limited access to their records, restricting the degree to which these markets can be effectively analyzed. In situations where the descriptive approach to market analysis fails, we turn instead to experimental economics and market simulation.

The goal of market simulation is to create a tightly controlled trading environment with configurable parameters and direct access to trading data. This environment is then exposed to traders, either human or computerized, and the market's behavior is observed. This is a much more flexible approach to the analysis of markets as it provides the ability to investigate any imaginable type of market, whether or not it has already been implemented in the real world. Additionally the tight control of input parameters and output variables allows researchers to make much stronger claims about causal relationships between market parameters and trading behavior.

The downside of the simulation approach to market analysis is that it can often be difficult to correctly emulate the behavior of traders. In the real world, the decision making process that controls traders' strategies is incredibly complex. Attempting to write software to copy this behavior gives approximate results at best. This approximation is sufficient for some types of market analysis, though sometimes a different approach is needed. When the subtleties of human decision making are a significant factor, we can instead invite human subjects to participate in our simulated market. This goes a ways towards accounting for human factors in trading environments and can produce simulation results which are arguably closer to expected real-world results. This type of simulation — where human subjects are used as traders — is the type of simulation which oTree Markets was built to produce.

2.2 Market Design at LEEPS Lab

LEEPS Lab has some experience with market simulations. An ongoing project at LEEPS is a simulation of high frequency trading, which seeks to examine the performance of various auction formats in high-speed algorithmic trading environments [1]. It does this through a complex market simulation that allows human players to control algorithmic trading bots. Another simulation experiment involves a complex heatmap-based trading interface [2]. This provides traders with an intuitive visual representation of the market state, theoretically allowing them to make trading decisions more quickly than an ordinary text interface would allow.

Both of these experiments were built from scratch, and they both quickly gained a large degree of design complexity. This complexity made them difficult to update and introduced a large potential for uncaught bugs. When more experimenters came to LEEPS desiring various types of market simulations, it became clear that there was a need for an extensible and simply-designed system for building these simulations that would allow for a wide variety of experiments while remaining easy to understand and debug. These are the conditions that led to the creation of the oTree Markets framework.

3 oTree

The central premise of experimental economics is that economic insights can be obtained by placing human subjects into carefully designed economic situations and observing their behavior. These experiments can take many forms, but the predominant technique is to implement experiments as video games. This is a convenient system as most experiments are easily implemented in this way, and computerization greatly simplifies the development and data collection processes. Various software solutions exist for the creation of these video game experiments, among them zTree [3] and LEEPS Lab's own ConG [4]. However, the current state of the art in economics experiment design is oTree, designed by Daniel Chen with programming assistance from Chris Wickens [5].

oTree experiments are built in Python, indirectly using the Django web framework. They are structured as web applications, where all of the experiment logic is written in Python and the interface is written in HTML/CSS. Experimenters create experiments by defining a series of pages, where each page shows players some information and collects some information from them in response. oTree contains many useful tools for the creation of these experiments including a powerful set of matching algorithms which allow fine-grained control over the way in which players are combined into groups and a large set of interface components allowing experimenters to create displays ranging from simple questionnaires to complex inter-

active games. oTree's has accessibility as a central design goal; it is intended to be simple enough to use that economics researchers with limited programming experience can quickly produce effective experiments without a steep learning curve.

While oTree's default configuration allows the creation of a wide variety of experiments, the sequential nature of its page system means that experiments that require direct, real-time interaction between players are not possible. Players can act on information submitted to previous pages, but communication inside a single page is not possible. Many types of experiments — including market simulations — depend on real-time interaction, so oTree requires an extension that adds support for this type of game. With the help of James Pettit, LEEPS Lab produced otree-redwood — an oTree extension that adds support for this real-time interaction to oTree. otree-redwood uses the Websocket protocol to add bidirectional communication between players, and adds a number of abstractions to the oTree API which simplifies the process of integrating real-time communication into oTree experiments. otree-redwood comes with an additional set of Javascript components which provide a convenient system for the creation of interfaces which utilize its real-time capability. LEEPS Lab has already used otree-redwood to create a number of real-time oTree experiments which were not previously possible to create. otree-redwood is a core component of oTree Markets, as market simulation is inherently a real-time process which requires that traders be able to communicate freely with the exchange. otree-redwood's simplicity and well-defined APIs have greatly eased the development of oTree Markets and will make its use much simpler.

4 Markets Background

Before discussing oTree Markets, we introduce some terminology which will be useful later when describing the design and function of oTree Markets' exchange component. The precise operation of an exchange is described by its market format. The market format encompasses the structure of orders in the market, as well as the algorithm used to calculate which orders transact with each other. In this section, we describe the Continuous Double Auction (CDA), a market format which is currently in use by most major financial exchanges. CDA's ubiquity made it the obvious choice for oTree Markets' exchange component, as it is likely that a large proportion of market simulation experiments will need to use this market format.

CDA, otherwise known as the consolidated limit order book, is defined by two components: an order structure and a trade calculation algorithm. CDA's orders are formally structured as "limit orders" with four major components:

1. Direction — buy (sometimes called a bid) or sell (sometimes called an ask or offer)
2. Limit quantity — maximum number of units to buy or sell
3. Limit price — highest acceptable price for a bid, lowest acceptable price for an offer
4. Time in force — indicating when the order should be canceled

For simplicity, oTree Markets' implementation of limit orders ignores the time in force field. This field is noncritical for many types of market experiment, so it was left out to reduce the exchange's complexity. If this feature were needed in the future, it would be a simple matter to update the messaging protocol and exchange implementation to support it.

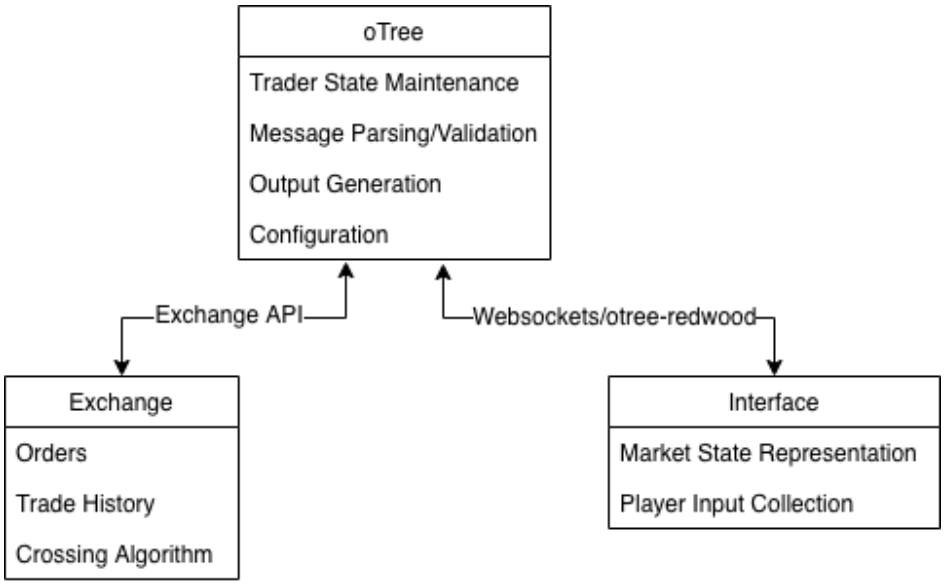
The exchange collects these limit orders as they are sent in by traders, first checking to see if their price is good enough to cross (trade) with any opposite-direction orders that are already in the exchange. If the new order does not cross it is simply added to the exchange's pool of orders, to be checked against each new incoming opposite-direction order.

To check an incoming order for crossings, its price is compared to the prices of the best opposite-direction orders in the exchange. For an incoming bid, this means that the exchange looks to see if its price is higher than the lowest-price ask in the market. For an incoming ask, it checks whether its price is lower than the highest-price bid. If this condition is met, then the order crosses and a trade occurs. This process is slightly complicated by orders with quantity greater than one, as the incoming order can transact with multiple other orders, and orders can be partially transacted. For a pseudocode description of oTree Markets' crossing algorithm for non-unit order volumes, see appendix A.

5 oTree Markets

The oTree Markets framwork has three core components:

Figure 1: oTree Markets block diagram



1. A continuous double auction exchange written in Python.
2. A reference interface implementation written in HTML/Javascript
3. An oTree experiment layer which connects and coordinates these components

Figure 1 provides a block diagram of this system. These three components are designed to be as independent as possible so that they can be swapped out for alternative versions when necessary. This modularity is one of the core design goals of oTree Markets. This system can be used as provided, but its intent is to be an easily understood and easily extended system for market simulation in oTree. It functions as a template for experiments, rather than as a final experiment itself.

5.1 Exchange Implementation

oTree Markets comes with one exchange implementation: a continuous double auction exchange with support for multiple-unit orders. This was chosen as the reference exchange implementation because of its ubiquity. Continuous double auction is a very common auction format and as such, many market simulation experiments will need an implementation of it.

Multiple-unit orders were chosen as they are more versatile and can be used with a wider variety of trading environments. However, this versatility comes at the cost of additional complexity. Multiple-unit orders are more complex as they can be partially filled, and so require a more complex crossing algorithm. Additionally, descriptions of trades become more complex as trades can now involve an arbitrary number of orders. This additional complexity will not be necessary for all market simulation experiments, but the added capability that this gives to the oTree markets framework is worth the sacrifice of simplicity.

A key requirement of oTree Markets' design is that no trading data is lost, and that it is possible to recover the market state from any point in the history of a trade session. To accomplish this, the exchange component is built as a series of database models. Django, the web framework which oTree is built on, comes with powerful tools for managing experiment data in a database. It uses a technique known as object-relational mapping to associate database entries with Python objects. This allows developers to create database models which exist both as Python objects and database entries.

The exchange implementation consists of three of these database models: an Exchange, an Order and a Trade. The Exchange model represents a market and contains all of the logic to enter orders and calculate the result of trades. Exchange models refer to a set

Figure 2: oTree Markets' reference text interface

of Order models which represent individual orders in a market. These Order models contain information such as the price and volume of each order. When a trade occurs, a Trade model is created which refers to the Orders which were involved in it. This data structure provides a convenient, intuitive representation of a market and allows easy access to the trading data from a session.

An important point to note about the order structure in oTree Markets is that orders always have integer price and volume. LEEPS' previous market experiments allowed orders to have floating point price and volume, which caused a number of bugs involving rounding errors and excessive precision. Integer price and volume means that all trade calculations can be done with integer arithmetic and that the possibility of rounding error is removed. This goes a long way towards preventing difficult-to-debug errors later in development.

5.2 oTree Layer

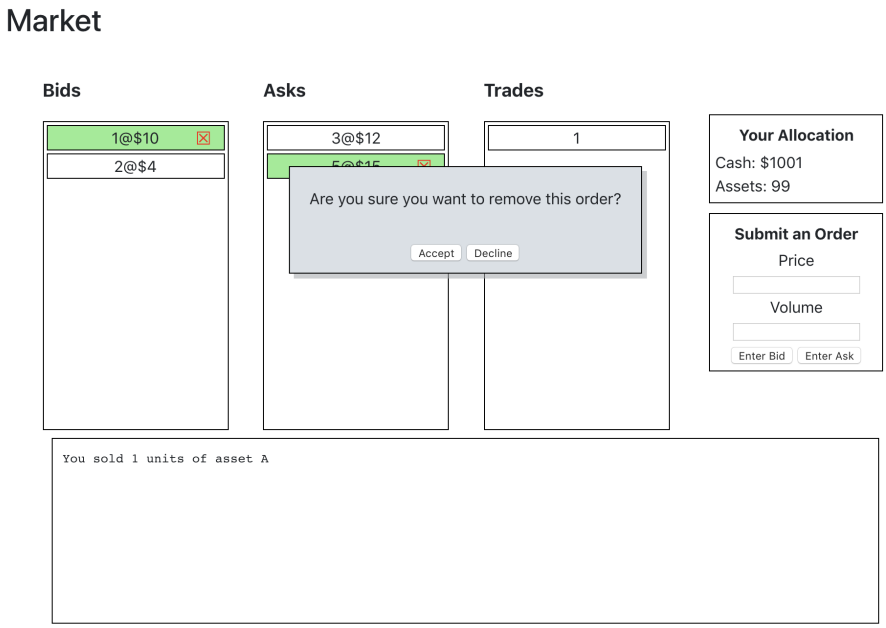
oTree markets uses oTree and otree-redwood as a communication layer between the exchange and frontend. This layer mainly deals with converting message formats to allow messages from the frontend to act on the exchange and vice versa. It also does several other tasks such as keeping track of each player's state and doing initial configuration before the round starts.

An essential function of this component is that is serves as oTree Markets' central source of truth. oTree maintains a complete image of the market and trader state and is responsible for managing this state and notifying other components when the state changes.

For example, when a player enters an order their interface is not immediately updated to reflect the newly entered order. The order message is sent to oTree which then passes it to the exchange. The exchange either accepts the order or a trade occurs. oTree takes this information from the exchange and updates its state appropriately. It then sends a state update message to the interface to inform the players that some change has occurred. This gives oTree complete control over acceptance or rejection of player inputs. This means that errors such as improperly formatted messages or orders sent by players with insufficient assets can be handled gracefully and without invalidating the state of the experiment.

The oTree layer also ensures that the interface conserves all of its data even when the page is refreshed. When the player's computer requests the in-

Figure 3: Text interface with confirmation modal



terface page, oTree collects all of the order and trade data which has been generated in the current session and passes it into Javascript so that it can be displayed as the current trading state. When the game first starts, this initial state is empty as no orders or trades have occurred. However if the player refreshes their page, this initial state will not be empty and will ensure that no inconsistencies arise between players' interfaces.

oTree is also responsible for generation and distribution of experiment output. Generating output is simple: oTree collects the Order and Trade model objects (discussed in section 5.1) which were created during the session and orders them sequentially. From this data, the market state at each point during the session is reconstructed and exported as a CSV table. This CSV data can then be analyzed using statistical packages, or any other desired technique.

5.3 Interface

The interface provided with oTree markets is meant as a simple reference implementation of a text-based trading interface. A screenshot of this interface is provided in figure 2. This interface allows players to manually enter the price and volume of their orders and displays a list of buy orders, a list of sell orders, and a list of trades that have occurred. Additionally it shows the player's current allocation of both cash and assets, and a message box with information such as errors and descriptions of completed trades. This interface provides a minimal amount of information and capability to players while still allowing them complete access to the market state and trading history. This interface is suitable for a wide variety of simple market simulation experiments, though more complex experiments would likely require modifications.

While this interface is meant to be minimal and basic, it is also designed with modularity in mind. It was built using the Polymer.js web framework which has modularity as a central design principle. Polymer.js is centered around the construction of "webcomponents" which are self-contained bundles of HTML and Javascript that serve as a single element of a webpage. For example, consider the section of the interface labeled "Bids". This is constructed as a webcomponent which takes a list of order data and represents it as a formatted list. It automatically detects this player's orders, colors them green and adds a red X which, when clicked, allows the player to cancel an already entered order. When an order is canceled, the interface first asks the player to confirm that they would like to cancel their or-

der by displaying a pop-up confirmation box (another self-contained webcomponent) as seen in figure 3. All of the individual components of the interface are reusable. Those desiring to create complex interfaces have these webcomponents as a starting point and will be saved from a large amount of redundant work.

6 Future Work

The future goals for the oTree Markets project are centered around expanding its capabilities to encompass a wider variety of simulated market environments. This includes expansions and modifications to the exchange component, alternate interfaces and the addition of automated trading bots.

One intended expansion is the implementation of alternate market formats such as frequent batch auctions, or exchange traded funds which would allow players to trade across multiple assets at once. The design of oTree Markets is such that implementing a replacement for the continuous double auction exchange which is currently included is a relatively simple task. The exchange has a simple API which, if followed correctly, should allow an alternative exchange to drop in with few modifications to other components.

Several alternate interfaces are in development. This includes a multiple-asset market where players are exposed to several different markets simultaneously, as well as an updated version of the visual trading interface discussed in section 2.2. The creation of alternate interfaces will also be a simple task as the messaging system is robust and well defined, and many components from the current interface will be reusable as mentioned in section 5.3.

Some experimenters have also expressed interest in adding automated traders to oTree Markets. This feature would enable experiments in which human subjects can trade against automated, algorithmic traders. This is useful as it would allow experimenters to artificially inflate the volume of orders in the market and to see how traders react to market conditions which would otherwise be difficult to produce. This is a longer-term goal, as it will require some restructuring of oTree Markets' messaging system, as well as the addition of a new API to allow experimenters to specify the behavior of these trading bots.

References

[1] Aldrich, Eric Mark and López Vargas, Kristian, Experiments in High-Frequency Trading: Comparing Two Market Institutions (February 21, 2019).

[2] Timothy N. Cason, Daniel Friedman, ED Hopkins, Cycles and Instability in a Rock–Paper–Scissors Population Game: A Continuous Time Experiment, The Review of Economic Studies, Volume 81, Issue 1, January 2014, Pages 112–136

[3] Fischbacher, U. z-Tree: Zurich toolbox for ready-made economic experiments. Exp Econ 10, 171–178 (2007)

[4] Pettit, J., Friedman, D., Kephart, C. et al. Software for continuous game experiments. Exp Econ 17, 631–648 (2014).

[5] Daniel L. Chen, Martin Schonger, Chris Wickens, oTree—An open-source platform for laboratory, online, and field experiments, Journal of Behavioral and Experimental Finance, Volume 9, 2016, Pages 88-97, ISSN 2214-6350,

A Calculating CDA crossings with non-unit order volumes

The below python-like pseudocode is the algorithm used to calculate order crossings in oTree Markets' exchange. This function is called when a new bid is entered into the exchange, either entering the new bid if its price is too low to cross with any asks, or calculating a trade if the price is high enough. This algorithm is given for bids, but the ask algorithm is very similar. The only differences are that an aggressive ask order is priced lower than the best bid and that incoming asks are compared with bids in descending order instead of ascending order.

```
def handle_incoming_bid(incoming_bid_order):
        # if incoming order isn't aggressive
        # enough just add it to the pool
        if incoming_bid_order < get_best_ask().price:
                add_to_order_pool(incoming_bid_order)
                return

        # keeps track of our order's remaining volume
        cur_volume = bid_order.volume
        # the list of orders which we've crossed with
        traded_orders = []
        for ask in get_asks_ascending():
                # stop when we hit an order that doesn't cross
                # or our order is out of volume
                if cur_volume == 0 or bid_order.price < ask.price:
                        break
                # if our remaining volume more than fills this order,
                # just update our remaining volume
                if cur_volume >= ask.volume:
                        cur_volume -= ask.volume
                # otherwise our remaining volume is 0 and the order we
                # trade with is partially filled and needs to be reentered
                else:
                        enter_new_ask(price=ask.price, volume=ask.volume - cur_volume)
                        ask.traded_volume = cur_volume
                        cur_volume = 0
                traded_orders.append(ask)
        # if our order has any volume left,
        # it was partially filled and needs to be reentered
        if cur_volume > 0:
                enter_new_bid(price=ask.price, volume=cur_volume)
        confirm_trade(taking_order=incoming_bid_order, making_orders=traded_orders)
```