BSE: The Bristol Stock Exchange

A minimal simulation of a limit order book financial exchange

Just enough for learning about basic trading algorithms

dc@cs.bris.ac.uk

1. Introduction

In very many human societies, for hundreds of years, buyers and sellers have met at marketplaces and haggled. When haggling, the seller states the offer-price that he or she wants to sell at, the buyer responds with a bid-price that is lower than the offer. The seller might drop the offer a little; the buyer might increase the bid a little; and they repeat these price revisions until they have struck a deal or one side gives up on the haggling and no deal occurs.

In the language of economics, the word "auction" is used to refer to the means by which buyers and sellers come together to exchange money for goods. There are lots of different types of auction.

One of the most famous types is the *English Auction*, where the seller stays silent and the buyers announce increasing bid-prices until only one buyer remains, who then gets the deal. This is a popular way of selling fine art, and livestock too. A more technical name for the English auction is a *first-price ascending-bid auction* because the first (highest) price becomes the transaction price. In contrast, if you've every bought anything on eBay, you'll know that a lot of the auctions there are run as *second-price:* that is, you win the deal by bidding highest, but the price you pay is the bid-price of the second-highest bidder. And, when you're in a shopping mall, you're in an auction too. It's what economists refer to as a *Posted Offer Auction*: the sellers name (or "post") their offer-prices, and the buyers simply take it or leave it at that price.

However, if you go to Amsterdam or Rotterdam and try to buy tulip or daffodil bulbs (a big business in the Netherlands) you'll see almost exactly the opposite process in action. In the *Dutch Flower Auction*, the buyers stay silent while the seller starts with an initial high offer-price and then gradually drops the offer price until a buyer jumps in to take the deal. This is a *descending-offer* auction.

In many the world's major financial markets, a style of auction is used which is a very close relative of the basic haggling process. This is much like having an ascending-bid and a descending-offer auction going on in one market at the same time. It is known as the *Continuous Double Auction*, or *CDA* for short. In the CDA, a buyer can announce a bid at any time and a seller can announce an offer at any time. While this is happening, any seller can accept any buyer's bid at any time; and any buyer can accept any seller's offer at any time. It's a continuous asynchronous process, and it needs no centralised auctioneer, but it does need some way of recording the bids and offers that have been made: this is the *limit order book* (LOB) that we will look at in some detail here.

The CDA interests economists because, even with a very small number of traders, the *transaction prices* (i.e. the agreed deal-prices) rapidly approach the theoretical market equilibrium price. The equilibrium price is the price that best matches the quantity demanded to the quantity supplied by the market, and in that sense it is the most efficient price for the market. The CDA is also of pragmatic interest because of the trillions of dollars that flow through national and international CDA-based markets in commodities, equities (stocks and shares), foreign exchange, fixed income (tradable debt contracts such as government bonds, known in the UK as *gilts* and in the USA as *treasury bills*), and derivatives contracts. Although there are still some exchanges where human traders physically meet in a central trading pit and shout out verbal bids and offers, in very many major markets the traders engage with one another remotely, via a screen-based electronic market, interacting by placing quotes for specific quantities at specific prices on the LOB.

The LOB displays data that summarises all the outstanding bids and offers, i.e. the "live" orders that have not yet cancelled by the traders that originated them. In market terminology, offers are also referred to as *asks*, and the LOB has two sides: the bid book and the ask book. The bid book shows the prices of outstanding bid orders, and the quantity available at each of those prices, in descending order, so that the best (highest) bid is at the top of the book. The ask book shows the prices of outstanding asks, and the quantity available at each of those prices, in ascending order, so that the best (lowest) ask is at the top of the book.

So, for example, if there are two traders each seeking to buy 30 shares in company XYZ for no more than $1.50 per share, and one trader hoping to buy 10 for a price of $1.52; and at the same time if there was one trader offering a 20 shares at $1.55 and another trader offering 50 shares at $1.62, the LOB for XYZ would look something like the one illustrated in Figure 1.1, and traders would speak of XYZ being priced at "152-55".

XYZ			
Bid		Ask	
10	152	155	20
60	150	162	50

Figure 1.1: how a limit order book (LOB) might be displayed on a trader's screen

The information shown on a LOB is sometimes referred to as "level 2" or "market depth" data. In contrast, "Level 1" market data just shows the price and size (quantity) for the best bid and ask, along with the price and size of the last recorded transaction, of the instrument being traded. Some people like to try their hand at "day trading" on their home PCs and they often operate with even more restricted data, such as simply whatever price the instrument was last traded at, or the mid-price, the point between the current best bid and the best ask (so in this example, the mid-price of XYZ would be $1.535). As level 2 data is what the pros use, that's what you get in BSE.

2. BSE: The Bristol Stock Exchange

The *Bristol Stock Exchange* (BSE) is a minimal simulation of a financial exchange running a limit order book (LOB) in a single tradable security. It abstracts away or simply ignores very many complexities that you find in a real financial exchange. In particular, a trader can at any time issue a new order, which replaces any previous order that the trader had on the LOB: that is, any one trader can have at most one order on a LOB at any one time. Furthermore, as currently configured, BSE assumes zero latency in communications between the traders and the exchange, and also (very conveniently) assumes that after any one trader issues an order that alters the LOB, the updated LOB is distributed to all traders (and, potentially, also results in a transaction) before any other trader can issue another order.

Be aware: real-world exchanges and markets are much more complicated than this.

BSE is written in Python. It's single-threaded and intended to be run in batch-mode, writing data to files for subsequent analysis, rather than single-stepping with dynamic updating of displays via an interactive graphical user interface (GUI). A GUIfied version is in the pipeline, but even that would need to switch into batch-mode to generate statistically reliable data from many hundreds or thousands of repeated market simulation experiments, as you'll see demonstrated in Section 5.

For ease of distribution, and to help people who are new to Python, currently everything is in one file: `BSE.py`, the latest major release of which is available for download from:

https://github.com/davecliff/BristolStockExchange/downloads

The latest version of this document should also be available from that GitHub repository, as `BSEguide.pdf`.

The output data-files created by BSE.py are all ASCII comma-separated values (i.e., files of type `.csv`) because that format can easily be imported by spreadsheet programs (such as OpenOffice *Calc*, Apple *Numbers*, or Microsoft *Excel*) and can also be readily imported by more sophisticated statistical analysis systems such as *R* or *Matlab*.

BSE.py has been written to be easy to understand; it is certainly not going to win any prizes for efficiency; probably not for elegance either. It's roughly 1000 lines of code, all in one long file rather than split across multiple files.

Examining the BSE code you can see that the Exchange has to keep internal records of exactly which trader submitted which order, so that the book-keeping can be done when two traders enter into a transaction, but that the LOB it "publishes" to the traders deliberately discards a lot of that data, to anonymize the identity of the traders. The `Exchange` class's `publish` method uses values from the exchange's internal data structures to build the market's LOB data as a Python dictionary structure containing the time, the bid LOB and the ask LOB:

```
def publish_lob(self, time):
                public_data={}
                public_data['time']=time
                public_data['bids']={'best':self.bids.best_price,
                                     'worst':self.bids.worstprice,
                                     'n': self.bids.n_orders,
                                     'lob':self.bids.lob_anon}
                public_data['asks']={'best':self.asks.best_price,
                                     'worst':self.bids.worstprice,
                                     'n': self.asks.n_orders,
                                     'lob':self.asks.lob_anon}
                return public_data
```

The bid and ask LOBs are both also dictionary structures. Each LOB shows:

- the current best price;
- the worst possible price (i.e. the lowest-allowable bid-price, or the highest-allowable offer price: these values can be of use to trader algorithms, i.e. for making "stub quotes": an example is given later in this document);
- the number of orders on the LOB;
- and then the anonymized LOB itself.

The anonymized LOB is a list structure, with the bids and asks each sorted into ascending order (so the best bid is the *last* item on the bid LOB, but the best ask is the *first* item on the ask LOB). Each item in the list is a pair of items: a price, and the number of shares bid or offered at that price. Prices for which there are currently no bids or asks are not shown on the LOB.

Orders that are issued by a trader have a trader-i.d. (TID) code, an order type (bid or ask), price, quantity, and a timestamp. NB in the current version of BSE, for extra simplicity, the quantity is always 1. For illustration, in this document we'll write the order as a list: [TID, type, price, quantity, time], hence if trader T22 bids $1.55 for one share at time t=10 seconds after the market opens, we'd write the order as:

```
[T22,bid,155,1,0010].
```

So, for example, if the bid and ask LOBs are initially blank and then the following sequence of orders is issued, the bid and ask LOBs will alter as shown:

```
[T11, bid, 022, 1, 0002] bid:[(22, 1)]         ask:[]
[T02, bid, 027, 1, 0006] bid:[(22, 1),(27, 1)] ask:[]
[T08, ask, 077, 1, 0007] bid:[(22, 1),(27, 1)] ask:[(77, 1)]
[T01, bid, 027, 1, 0010] bid:[(22, 1),(27, 2)] ask:[(77, 1)]
[T03, ask, 062, 1, 0018] bid:[(22, 1),(27, 2)] ask:[(62, 1),(77, 1)]
[T11, bid, 030, 1, 0021] bid:[(27, 2),(30, 1)] ask:[(62, 1),(77, 1)]
```

Note that the order issued at t=21 comes from trader T11, and hence replaces T11's previous order issued at t=2, which is why the bid at $0.22 is deleted from the bid LOB at t=21. At this stage, the *bid-ask spread* (i.e., the difference between the best bid and the best ask) is $0.32.

In financial-market terminology, if a trader wants to sell at the current best bid price, that's referred to as "hitting the bid"; if a trader wants to buy at the current best ask-price, that's referred to as "lifting the ask". Both are instances of "crossing the spread". So, to continue our example, let's say that trader T02 decides to lift the ask. In BSE, this is signalled by the trader issuing an order that crosses the spread, i.e. issuing a bid

priced at more than the current best ask, and the transaction then goes through at whatever the best price was on the LOB as the crossing order was issued. So, if at *t*=25s T02 lifts the ask by bidding $0.67, we'd see this:

```
[T02, bid, 067, 1, 0025] bid:[(27, 1),(30, 1)] ask:[(77, 1)]
```

Note that T02 already had an earlier bid on the LOB, one of the two priced at 27, so when T02's new bid of 67 was received by the exchange, it first deleted the earlier bid (i.e., the new order replaced the old one), and then the exchange system detected that the bid crossed the best ask (which was priced at 62, from the order sent at *t*=18 by trader T03) and so the exchange deleted that ask from the LOB too. Immediately afterwards, the exchange and the traders do some book-keeping: the exchange records the transaction price and time on its "tape" (the time-series record of orders on the exchange); and the two counterparties to the trade, traders T02 and T03, each update their "blotters" (their local record of trades they have entered into) -- these uses of "tape" and "blotter" are common terminology in the markets.

Now let's look at some simple automated trading algorithms, or "robot traders". The simple algorithms explored here are all automated execution systems: that is, they automate the process of executing an order that has originated elsewhere. In investment banking, the human workers that do this job are known as "sales traders": a human sales trader waits for an order to arrive from a customer, and then works that order in the market: that is, the sales trader just executes the order without giving any advice or comment on the customer's decision to buy or sell. At its simplest, a customer order will state what instrument (e.g., which stocks or shares) the customer wants to deal in, whether she wants to buy or sell, how many, and what price she wants. If the customer is keen to complete the transaction as soon as possible, she should specify a *market order*, i.e. just do the deal at whatever is the best price in the market right (i.e., on the LOB) now. But if the customer is happy to wait, then she can specify a *limit order*, a maximum price for a purchase or a minimum price for a sale, and then the sales trader's job is to wait until the conditions are right for the deal to be done. The execution of limit orders is where the sales trader can make some money. Say for example that the customer has sent an order stating that she wants to sell one share of company XYZ for no less than $10.00, at a time when XYZ is trading at $9.90 but where the price has been rising steadily: if the sales trader waits a while and executes the order when XYZ has risen to $10.50, the trader can return the $10.00 to the customer and take the extra $0.50 as a fee on the transaction; but if the trader instead executes this order at the precise moment that the price of XYZ hits $10.00, then the customer's order has been satisfied but there is no extra money, no margin on the deal, for the trader to take a share of.

So, the simple robot traders we'll look at here can all be thought of as computerized sales-traders: they take customer limit orders, and do their best to execute the order in the market at a price that is better than the limit-price provided by the customer (this is the price below which they should not sell, or above which they should not buy). Customer orders are issued to traders in BSE and then the traders issue their own orders as bids or asks into the market, trying to get a better price than the limit-price specified by the customer.

The core of a market session in BSE is a `while` loop that repeats once per time-step. In each cycle of this loop, the following things happen:

- There is a check to see if any new customer orders need to be distributed to any of the traders, via a call to `customer_orders()`.
- An individual trader is chosen at random to issue its current response by calling the trader's `getorder()`method: this will either return the value `None`, signaling that the trader is not issuing a order at the current time, or it will return an order to be added to the LOB; if that is the case then BSE processes the order via `exchange.process_order()`.
- If processing the order resulted in a trade, the traders do the necessary book-keeping, updating their blotters, via a call to `bookkeep()`.
- Each trader is given the chance to update its internal values that affect its trading behavior, via a call to the trader's `respond()` method.

Before any of that happens, the market needs to be populated by a call to `populate_market()` – that is where the number of traders and the type of each trader is determined. So the core loop of the BSE code looks roughly like this:

```
Traders = {}
populate_market(n_traders, traders)

while time < endtime:

	# how much time left, as a percentage?
	time_left = (endtime - time) / float(endtime - starttime)

	# distribute any new customer orders to the traders
	customer_orders(time, traders, order_schedule)

	# get an order (or None) from a randomly chosen trader
	tid = random_tid(traders)
	order = traders[tid].getorder(time,time_left,exchange.publish_lob(time))

	if order != None:

		# send order to exchange
		trade = exchange.process_order(time, order)

		lob = exchange.publish_lob(time)

		if trade != None:
			# trade occurred,
			# so counterparties update order lists and blotters
			traders[trade['party1']].bookkeep(trade, order)
			traders[trade['party2']].bookkeep(trade, order)
			trade_stats(expid, traders, tdump, time, lob)

	# traders respond to whatever happened
	lob = exchange.publish_lob(time)
	for t in traders:
		traders[t].respond(time, trade, lob)

	time = time + timestep
```

3. Experimenting with trading robots

3.1 Robot 1: Giveaway

The simplest robot trader in BSE is called *Giveaway*. Giveaway seemingly has no hope of making any money for itself because it just tries to execute the customer's order at exactly the limit price specified by the customer. Here is the code for Giveaway:

```
# Trader subclass Giveaway
# just give the deal away (but never makes a loss)
class Trader_Giveaway(Trader):

        def getorder(self, time, countdown, lob):
                if len(self.orders) < 1:
                        order = None
                else:
                        quoteprice = self.orders[0].price
                        self.lastquote = quoteprice
                        order = Order(self.tid, self.orders[0].otype,
                                      quoteprice, self.orders[0].qty, time)
                return order
```

Giveaway is really dumb, but it does serve to illustrate the core of what it takes to make a trader robot in BSE. A trader robot needs to define two core functions: `getorder()` and `respond()`. The `getorder()` method specifies how the robot calculates the price of its next order in the market, and `respond()` specifies how it alters the values of any variables used in `getorder()`, in response to events in the market. Giveaway is so simple that it doesn't need to define its own `respond()` method; it just inherits the default do-nothing method from its `Trader` superclass. Giveaway's `getorder()` is also very simple: it checks to see that it actually has a customer order to execute; if it doesn't, then it returns an order of `None`; otherwise it creates an order to be sent to the exchange, using the price copied directly from its current customer order.

We can run an experiment with a market populated just by Giveaway traders. To do that, we'd edit `populate_market()` so that all the traders are instances of Giveaway, and we would have to specify a schedule for the arrival of customer orders, by editing `customer_orders()`. Then we would call `market_session()` to run the market for some period of time.

In all the example market sessions shown in this section, we'll use a version of `customer_orders()` that assigns one order to each of 62 traders (31 of which are working buy orders, and 31 are working sell orders) at time t=0, and then no new orders arrive until t=30 (i.e., after 30 simulated seconds) at which point all traders' previous customer orders are cancelled and each trader is re-assigned one fresh order. A similar replenishment happens every 30 seconds until three minutes have passed, at which point the market session ends. The customer order prices are structured so that the market's *equilibrium price* (the price at which supply and demand are balanced) is $1.00. Further, more detailed discussion of `customer_orders()` comes later, in Section 4.

At the end of the session, the file written by the exchange's `tape_dump()` shows the history of transactions in the market, as illustrated in Figure 3.1.1; and the file written by `trade_stats()` shows the average aggregate profit of the different types of traders

in the market: as there is only one type of trader, there is only one line on that graph, as shown in Figure 3.1.2.

Now, as Giveaway never aims to make any profit, we might expect that the line in Figure 3.1.2 should be flat at zero, but it isn't. To understand why this is so, consider this example: if trader T1 is working a sell order and offers at its limit price of $10 and trader T2 has a buy order with a limit price of $12, then when T2 issues its bid at $12, that bid crosses T1's offer and so the trade executes at a price of $10, generating a $2 profit for T2. That is, in a market populated solely by Giveaway robots, some of the traders end up making money simply by luck, because there are other traders in the market with different valuations, giving away the deal.

Note that in a lot of academic publications on trading robots, including those cited in this document, researchers have monitored market-level statistics such as *Smith's* α which is a measure of how well the traders in the market collectively 'discover' the underlying equilibrium price. Here we are only monitoring trader's average profit from working the customer orders, as that is the kind of metric that matters at trading institutions such as banks or fund-management companies (and anyhow, in real markets, the equilibrium price is not known in advance and its value frequently alters, so any measures of equilibrium-finding make much less sense).

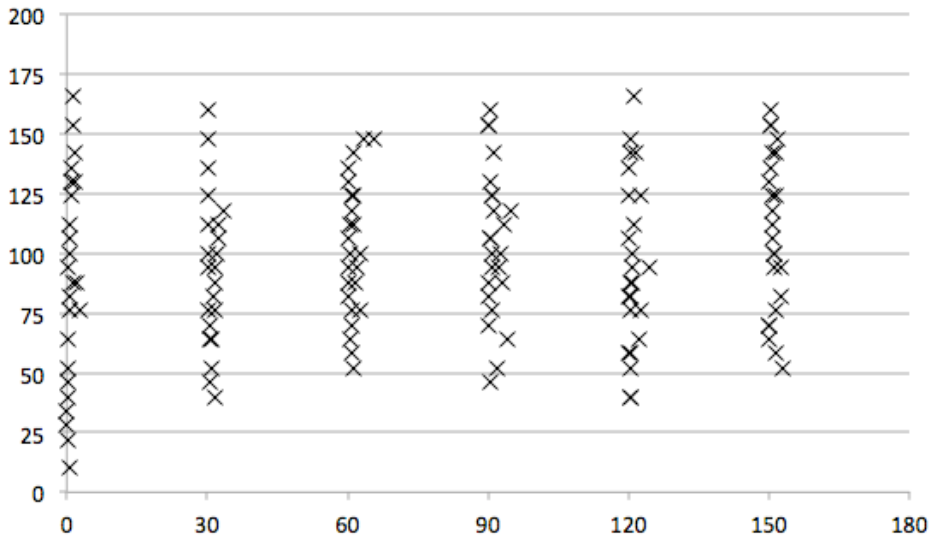

Figure 3.1.1: Transaction history for market populated by Giveaway traders. Vertical axis is price (in cents), horizontal axis is time (in seconds). The market session lasts 180s, with full simultaneous replenishment of orders every 30s. After each replenishment, there is a quick burst of trading activity that consumes most or all of the orders in the market.

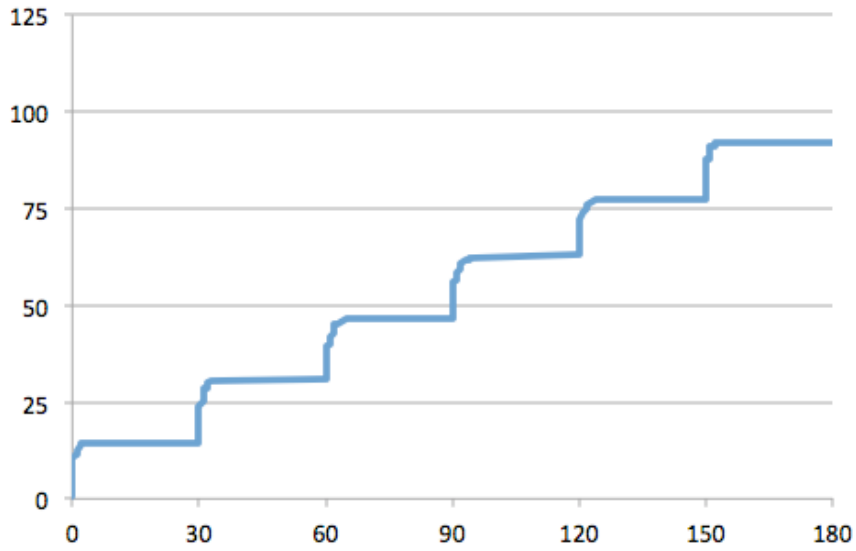

Figure 3.1.2: Average profit per Giveaway trader over 180s of trading. Horizontal axis is time; vertical axis is average profit accumulated per trader.

3.2 Robot 2: ZIC

Let's look now at a second type of trader-robot, one that might actually make some money. This is called ZIC, which stands for Zero-Intelligence Constrained, and was introduced by the economists Gode & Sunder (1993). As is hinted at in the name, ZIC traders have no intelligence at all, they just make up random prices for their quotes, but they are constrained to never make up a random price that would lead to a loss-making deal. That is, the customer's limit-price acts as a bound on the distribution from which the random prices are generated.

Here is the code for ZIC:

```
# Trader subclass ZI-C
# After Gode & Sunder 1993
class Trader_ZIC(Trader):

        def getorder(self, time, countdown, lob):
                if len(self.orders) < 1:
                        order = None
                else:
                        minprice = lob['bids']['best']
                        maxprice = lob['asks']['worst']
                        limit = self.orders[0].price
                        otype = self.orders[0].otype
                        if otype == 'Bid':
                                quoteprice = random.randint(minprice,limit)
                        else:
                                quoteprice = random.randint(limit,maxprice)

                        order = Order(self.tid, otype, quoteprice,
                                      self.orders[0].qty, time)
                return order
```

If we edit `populate_market()` so that all the traders are instances of ZIC, and run the same type of market session as we did for Giveaway, we get a similar-looking transaction time-series on the exchange tape, as shown in Figure 3.2.1; but now our robots are making more money as is shown on the profit graph in Figure 3.2.2.

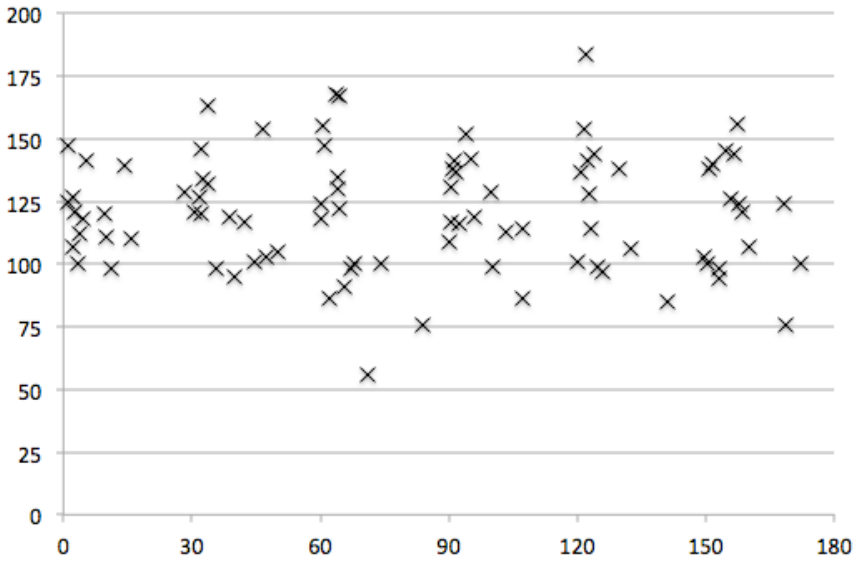

Figure 3.2.1: Transaction history for market populated by ZIC traders, over 180s of trading.

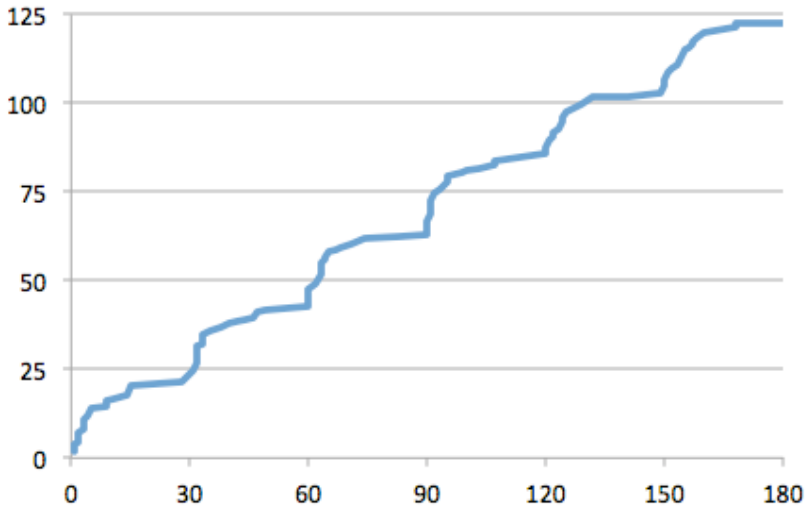

Figure 3.2.2: Average profit per ZIC trader over 180s of trading

3.3 Robot 3: Shaver

ZIC makes some money, but it doesn't use any information in the LOB. Our third type of robot, called *Shaver*, always tries to have the best bid or offer on the LOB, and does so by shaving a penny off the best price (so long as it can do so without violating the customer's limit price). Here is the code for Shaver:

```
# Trader subclass Shaver
# shaves a penny off the best price
# if there is no best price, creates "stub quote" at system max/min
class Trader_Shaver(Trader):

        def getorder(self, time, countdown, lob):
                if len(self.orders) < 1:
                        order = None
                else:
                        limitprice = self.orders[0].price
                        otype = self.orders[0].otype
                        if otype == 'Bid':
                                if lob['bids']['n'] > 0:
                                        quoteprice = lob['bids']['best'] + 1
                                        if quoteprice > limitprice :
                                                quoteprice = limitprice
                                else:
                                        quoteprice = lob['bids']['worst']
                        else:
                                if lob['asks']['n'] > 0:
                                        quoteprice = lob['asks']['best'] - 1
                                        if quoteprice < limitprice:
                                                quoteprice = limitprice
                                else:
                                        quoteprice = lob['asks']['worst']
                        self.lastquote = quoteprice
                        order = Order(self.tid, otype, quoteprice,
                                      self.orders[0].qty, time)

                return order
```

If we run a market session with a market populated entirely by Shavers, the kind of transaction histories and profit sequence are as illustrated in Figures 3.3.1 and 3.3.2.

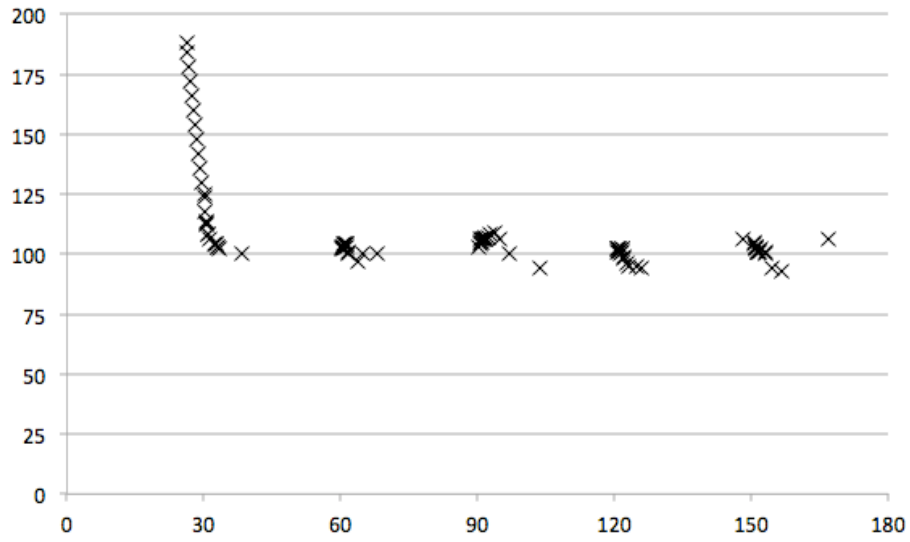

Figure 3.3.1: Transaction history for market populated by 62 Shaver traders, over 180s of trading. Initially, there are no trades for the first c.25 seconds while each of the shaver robots repeatedly makes a minimal improvement on the best bid or offer. The best bid rises from $0.01 to c.$1.90 and then holds at that level, set by the highest limit price on a buy order. Meanwhile the best offer is gradually being reduced but takes longer to fall from the system maximum (an arbitrarily high value: here it was $10.00) down to $1.90. At that point, transactions start to occur, initially at $1.90 but then at lower prices as the higher-priced limit orders are executed. Thereafter, the spread between best bid and best offer is always very small, and centered on the equilibrium price of $1.00.

.

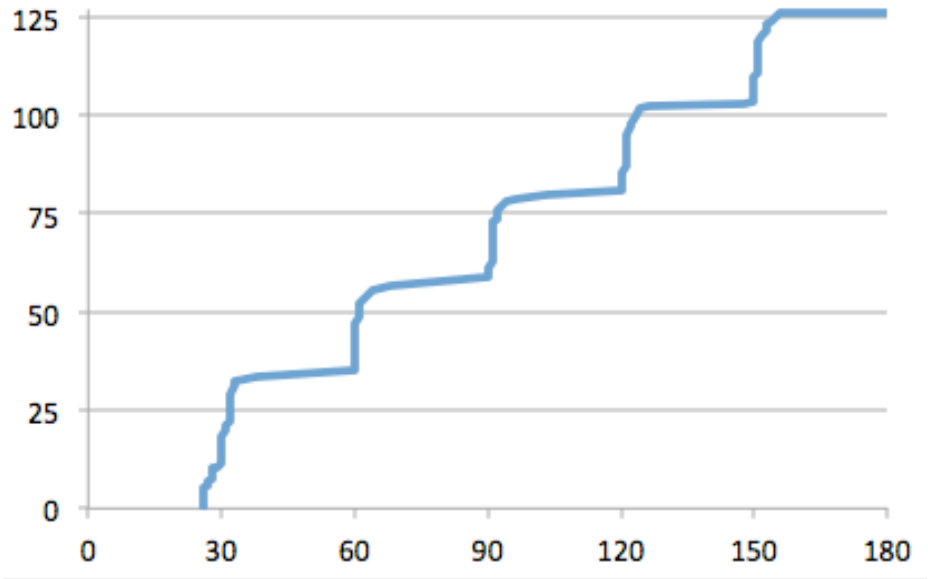

Figure 3.3.2: Average profit per Shaver trader over 180s of trading

3.4 Robot 4: Sniper

Shaver robots react to prices in the LOB, but they have no sense of urgency: they steadily drop their prices by a single penny each time. If the market stays open forever then eventually a Shaver might drop or raise its price to the point where it gets a trade, but if the market is closing soon then it seems pretty obvious shaving by more than a penny might increase the chance of getting a trade. In fact, if there are only a few seconds left before the market closes, it might be best to simply cross the spread to get a trade rather than get no trade at all.

In a famous early public contest in automated trading on experimental markets, organized at the Santa Fe Institute, Todd Kaplan submitted a trader-robot that (in essence) lurked in the market doing very little and then, just before the market closed, came in to the market to "steal the deal". This strategy, now known as *Kaplan's Sniper*, won the contest: see Rust *et al.* (1992) for further details. Here, we'll edit the *Shaver* to include elementary sniping capability: our version of Sniper uses the `getorder()` parameter `countdown`, which should be the percentage of time remaining in the market session, to lurk for a while and then rapidly increase the amount shaved off the best price as time runs out. The code is below, with underlining to highlight what's been altered:

```
# Trader subclass Sniper
# a Shaver that "lurks" until time remaining < threshold% of trading session
# gets increasingly aggressive, increasing "shave thickness" as time runs out
class Trader_Shaver(Trader):

        def getorder(self, time, countdown, lob):
                lurk_threshold = 0.2
                shavegrowthrate = 3
                shave = 1.0/(0.01 + countdown/(shavegrowthrate*lurk_threshold))
                if (len(self.orders) < 1) or (countdown > lurk_threshold):
                        order = None
                else:
                        limitprice = self.orders[0].price
                        otype = self.orders[0].otype
                        if otype == 'Bid':
                                if lob['bids']['n'] > 0:
                                        quoteprice = lob['bids']['best'] + shave
                                        if quoteprice > limitprice :
                                                quoteprice = limitprice
                                else:
                                        quoteprice = lob['bids']['worst']
                        else:
                                if lob['asks']['n'] > 0:
                                        quoteprice = lob['asks']['best'] - shave
                                        if quoteprice < limitprice:
                                                quoteprice = limitprice
                                else:
                                        quoteprice = lob['asks']['worst']
                        self.lastquote = quoteprice
                        order = Order(self.tid, otype, quoteprice,
                                      self.orders[0].qty, time)

                return order
```

The transaction-price and profit time-series in a market populated entirely by our Sniper robots are quite different from those seen in the previous three types of robot. Predictably, nothing happens until quite close to the end of the market, as is seen in Figure 3.4.1. Profits are shown in Figure 3.4.2.

It is well known that a market populated entirely by sniper robots shows these kind of dynamics: snipers are essentially parasitic; they can do very well, so long as there are other types of traders present in the markets engaging in competitive price-discovery. Put most simply, snipers rely on "stealing the deal" from other traders, and this can work out so long as all those other traders are not also running a sniper strategy.

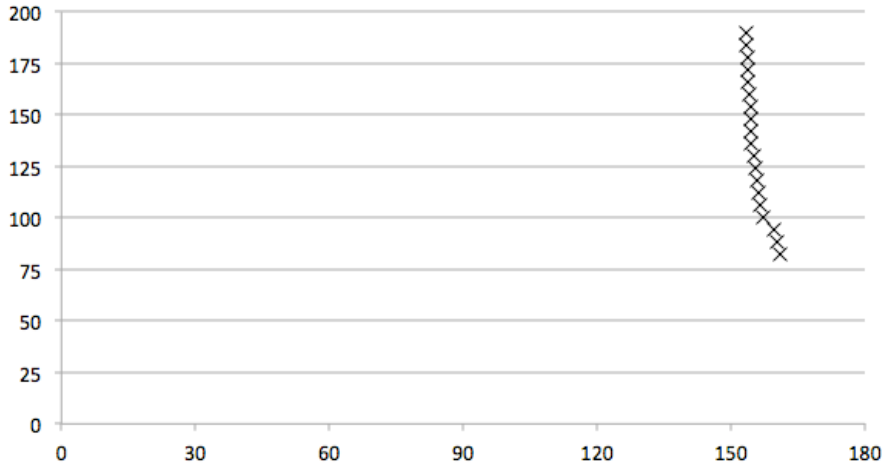

Figure 3.4.1: Transaction history for market populated by Sniper traders, over 180s of trading. There is a long period where each sniper is "lurking", simply observing the market, and then once the time remaining until the close of the session has fallen to a low enough value, the snipers become active, generating a transaction-price time series similar to that of the "shaver" robots.

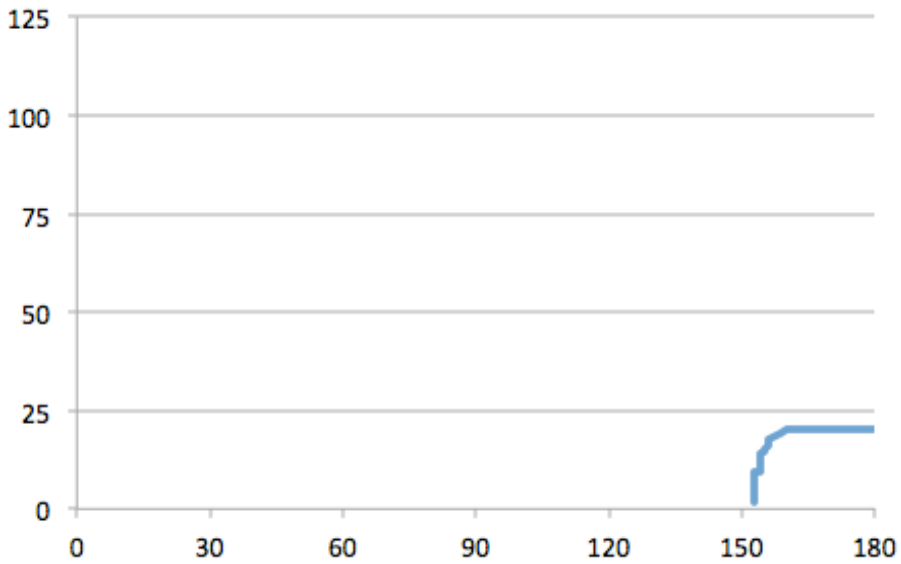

Figure 3.4.2: Average profit per Sniper trader over 180s of trading.

3.5 Robot 5: ZIP

The Zero-Intelligence Plus (ZIP) trading algorithm was invented in 1996 by Cliff, and the original C source code for the reference implementation was released as part of a technical report published as (Cliff, 1997). The version of ZIP in BSE closely follows the structure of the C reference implementation, for ease of comparison, and so is perhaps not as pythonic as a fresh re-implementation would be. The parameter values used to initialize ZIP here are based on those originally published in (Cliff, 1997). In 2001, Das *et al*. at IBM reported on experiments in which they demonstrated that ZIP, and also IBM's "MGD" algorithm, could outperform human traders.

The `getorder()` for ZIP is quite straightforward:

```
def getorder(self, time, countdown, lob):
                if len(self.orders) < 1:
                        self.active = False
                        order = None
                else:
                        self.active = True
                        self.limit = self.orders[0].price
                        self.job = self.orders[0].otype
                        if self.job == 'Bid':
                                # currently a buyer (working a bid order)
                                self.margin = self.margin_buy
                        else:
                                # currently a seller (working a sell order)
                                self.margin = self.margin_sell
                        quoteprice = int(self.limit * (1 + self.margin))
                        self.price = quoteprice
                        order=Order(self.tid, self.job, quoteprice,
                                    self.orders[0].qty, time)

                return order
```

As can be seen, ZIP generates a price to quote by multiplying the limit price specified on the customer's order by 1+`margin`. For sell orders, `margin` (which is referred to as $\mu(t)$ in Cliff, 1997) is positive, while for buy orders it is negative. The key innovation in ZIP was to use machine learning techniques to alter the value of `margin` in response to events in the market: for that reason, ZIP is the first of the five trading robots introduced here to override the default do-nothing `respond()` method. ZIP's `respond()`is quite a lengthy method, and for a full explanation of why it is as it is, see Cliff (1997). The original 1997 reference implementation of ZIP was studied in simulated markets without a LOB: instead ZIP just responded to whatever happened to the previous quote in the market; this set-up was based on the pioneering -- and subsequently Nobel-prize-winning -- experiments of Vernon Smith (see, e.g., Smith, 1962). The implementation here, for ZIP operating on the LOB-based BSE, requires some extra code and extra internal variables so that a ZIP trader can react to changes in the LOB. The code for ZIP dealing with the bid-LOB is shown below; the code for the ask-LOB is much the same, with appropriate changes.

```
# what, if anything, has happened on the bid LOB?
	bid_improved = False
	bid_hit = False
	lob_best_bid_p = lob['bids']['best']
	lob_best_bid_q = None
	if lob_best_bid_p != None:
		# non-empty bid LOB
		lob_best_bid_q = lob['bids']['lob'][-1][1]
		if self.prev_best_bid_p < lob_best_bid_p :
			# best bid has improved
			# NB doesn't check if the improvement was by self
			bid_improved = True
		elif trade != None
		     and ((self.prev_best_bid_p > lob_best_bid_p)
		     or  ((self.prev_best_bid_p == lob_best_bid_p)
		      and (self.prev_best_bid_q > lob_best_bid_q))):
			# previous best bid was hit
			bid_hit = True
	elif self.prev_best_bid_p != None:
		# the bid LOB has been emptied by a hit
		# NB works cos current BSE does not allow order-cancels
		bid_hit = True
```

So once the bid and ask LOBs have been examined, the values of the Boolean variables `bid_improved`, `ask_improved`, `bid_hit`, and `ask_lifted` reflect the ZIP trader's view of what's happened in the LOB. The trader's response is then determined by a simple "decision tree", a set of nested `if-then` conditions, that assign a value to `target_price`, the price that the ZIP trader is "aiming for": again, we'll just look at the code for when the ZIP is working a bid order; the code for working an ask is very similar:

```
deal =  bid_hit or ask_lifted

if self.job == 'Bid':
	# buyer
	if deal :
		tradeprice = trade['price']
		if self.price >= tradeprice:
			# could buy for less? raise margin (i.e. cut the price)
			target_price=target_down(tradeprice)
			profit_alter(target_price)
		elif bid_hit and self.active and not willing_to_trade(tradeprice):
			# wouldn't have got this deal, still working order,
			# so reduce margin
			target_price=target_up(tradeprice)
			profit_alter(target_price)
	else:
	# no deal: aim for a target price lower than best ask
		if bid_improved and self.price < lob_best_bid_p:
			if lob_best_bid_p != None:
				target_price = target_down(lob_best_ask_p)
			else:
				target_price = lob['bids']['worst'] #stub quote
			profit_alter(target_price)
```

The function `willing_to_trade()` is pretty straightforward:

```
def willing_to_trade(price):
        # am I willing to trade at this price?
        willing = False
        if self.job == 'Bid' and self.active and self.price >= price:
        willing = True
        if self.job == 'Ask' and self.active and self.price <= price:
                willing = True
       return willing
```

The value of `target_price`, is set by a call to one of `target_down()` or `target_up()`, depending on what has happened on the LOB, and what type of order the trader is working. These two functions introduce some random noise into the process, limited by the trader's `cr` and `ca` parameters:

```
def target_up(price):
        # generate a higher target price by randomly perturbing given price
        ptrb_abs = self.ca * random.random() # absolute shift
        ptrb_rel = price * (1.0 + (self.cr * random.random()) ) # relative shift
        target=int(round(ptrb_rel + ptrb_abs,0))
        return(target)

def target_down(price):
        # generate a lower target price by randomly perturbing given price
        ptrb_abs = self.ca * random.random() # absolute shift
        ptrb_rel = price * (1.0 - (self.cr * random.random()) ) # relative shift
        target=int(round(ptrb_rel - ptrb_abs,0))
        return(target)
```

Finally, once the target price has been set, a ZIP trader alters its profit margin via the `profit_alter()` method:

```
def profit_alter(price):
        oldprice = self.price
        diff = price - oldprice
        change = ((1.0-self.momntm)*(self.beta*diff))
                 + (self.momntm*self.prev_change)
        self.prev_change = change
        newmargin = ((self.price + change)/self.limit) - 1.0

        if self.job=='Bid':
            if newmargin < 0.0 :
                    self.margin_buy = newmargin
            self.margin = newmargin
        else :
            if newmargin > 0.0 :
                    self.margin_sell = newmargin
            self.margin = newmargin

        #set the price from limit and profit-margin
        self.price = int(round(self.limit*(1.0+self.margin),0))
```

And that is all that a ZIP trader involves.

Figure 3.5.1 shows a transaction history: clearly, a market populated entirely by ZIP traders can rapidly converges to a stable state where transactions are all talking place close to the market's equilibrium price of $1.00. Figure 3.5.2 shows the average profit of the ZIP traders: it would seem that in this particular test situation, ZIP does better than Sniper but is outperformed by Giveaway, ZIC, and Shaver.

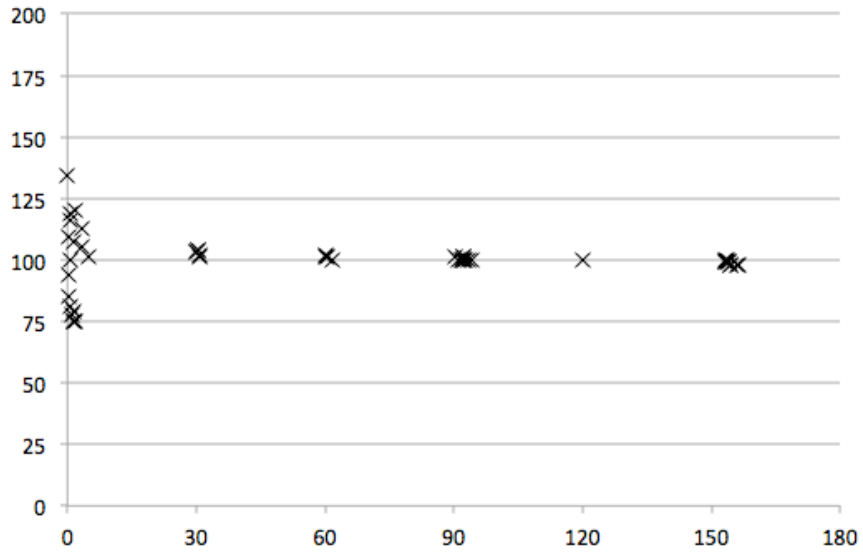

Figure 3.5.1: Transaction history for market populated by ZIP traders, over 180s of trading. Markets populated by ZIP traders can rapidly converge on the equilibrium price (here $1.00).

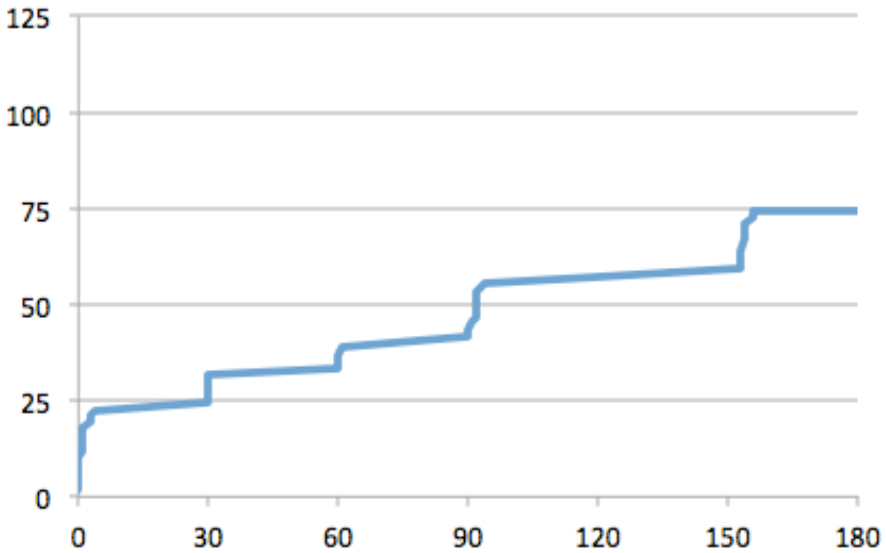

Figure 3.5.2: Average profit per ZIP trader over 180s of trading: although ZIP is good at finding the equilibrium, it makes less average profit per trader than some other simple strategies.

4. Altering the market supply and demand schedules in BSE

The flow of customer orders into the market is determined by the markets' supply schedule (for sell orders) and demand schedule (for bid orders). These are specified in the `order_schedule` argument to the main `market_session()` method. Within BSE, the `customer_orders()` method reads the supply and demand schedules, and any additional parameters, and assigns customer orders to the schedules.

In his seminal market-trading experiments, Vernon Smith (1962) used the approach of simultaneously assigning orders to all the (human) traders in the experiment, and then allowing them to trade for a period that ended either when a time-limit (typically of around 10 minutes) was reached, or when no more orders could be executed, and then that session (or "trading day" as Smith called them) would be ended and a new batch of orders could be distributed to all traders, to initiate the next session. Smith's early experiments typically involved five or six rounds of such *simultaneous replenishment*. In later experiments, Smith and his colleagues explored the effects of switching to *continuous replenishment* where the orders are trickle-fed into the market, with different traders receiving new orders at different times, rather than everyone getting new orders at the same time (see Cliff & Preist, 2001, for further discussion).

Here we'll look first at how to configure `order_schedule()` so that `customer_orders()` generates periodic simultaneous replenishment, as that's the style of replenishment that was first used by Vernon Smith in his early experiments, and has since been used by many other experimenters. Say, for example, that we want to run an experiment lasting for 180 seconds (i.e., 3 minutes), where we replenish fresh orders to all traders once every 30 seconds, and that all orders, sell or buy, should be priced in the range $0.90-$1.10 with fixed spacing of orders between the minimum and the maximum (we'll see exactly what that means, further below); we'd express that as follows:

```
start = 0.0
end = 180.0
range = (10, 190)
supply_sched = [{'from':start, 'to':end, 'ranges':[range], 'stepmode':'fixed'}]
demand_sched = [{'from':start, 'to':end, 'ranges':[range], 'stepmode':'fixed'}]
order_schedule = {'sup':supply_sched, 'dem':demand_sched,
                  'interval':30, 'timemode'='periodic'}
```

Code Snippet 4.1: setting up supply and demand to be symmetric (use the same schedule range and step-size), and with periodic replenishment of orders, once every 30 seconds.

The value of `stepmode` in the schedules tells BSE how to generate the prices of orders across the specified range: the possible values for `stepmode` are as follows:

- `'fixed'`: prices are generated to give a constant difference between successive prices when sorted into order. This gives a regular stepped appearance to the supply and demand curves in the market, as shown in Figure 4.3. The supply and demand curves are explained in more detail in the text immediately below.

- `'random'`: each order price is generated at random from a uniform distribution specified by the schedule's `ranges` values. The supply and demand curves that result from using random mode can vary quite widely, from one call to the next, as illustrated in Figure 4.4.

- `'jittered'`: order prices are generated with regular intervals, as in `'fixed'` mode, but with some small random noise added in to break up the regularity, as illustrated in Figure 4.5. The distribution of the noise is uniform over a range equal to the size of a single step if `stepmode` was `'fixed'`.

To understand how the schedules determine the supply and demand curves for the market, let's take a very simple example where there are six traders, named T1 to T6, working buy orders for units priced at \$1.50, \$1.65, \$1.80, \$1.95, \$2.10 and \$2.25 respectively. If all the sellers in the market were offering at \$2.30, only buyer T6 would be able to bid (because T6's limit price is \$2.25) and so the *quantity demanded* at \$2.30 is 1. In fact, the quantity demanded is 1 for any price sufficiently high that only T6 can bid: so any price p where $2.10 < p \leq 2.25$. At prices above \$2.25, none of the traders can bid, and so the quantity demanded is zero. But if all the sellers were offering at \$2.00, then two buyers (T5 and T6) would be able to bid, and so the quantity demanded at \$2.00 is 2. We can carry this on down to the situation where all the sellers are offering at \$1.50, and the quantity demanded at \$1.50 will be 6 because all the buyers are able to transact at that price. For any price lower than \$1.50, the quantity demanded will still be 6, because there are only six buyers in the market. We can draw this relationship between price and quantity demanded as a line on a graph, the *demand curve*, as illustrated in Figure 4.1.

In exactly the same way, the array of limit-prices on sellers' orders determines a supply curve for the market. Let's add in five more traders, T7 to T11, who are working sell orders with limit prices \$1.25, \$1.35, \$1.45, \$1.55, and \$1.65. The point where the supply and demand curves intersect determines the market's equilibrium point, with its corresponding equilibrium price and equilibrium quantity, as illustrated in Figure 4.2. The equilibrium point is important because it is where supply and demand are balanced by competition among the buyers and sellers: that is, it is a competitive equilibrium.

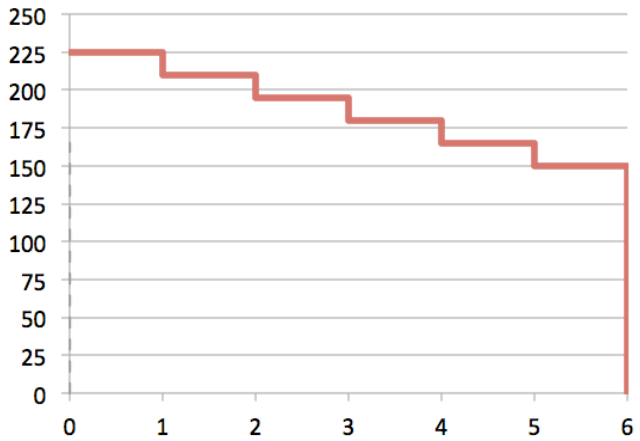

Figure 4.1: demand curve for six buyers. Horizontal axis is quantity; vertical axis is price. See text for discussion.

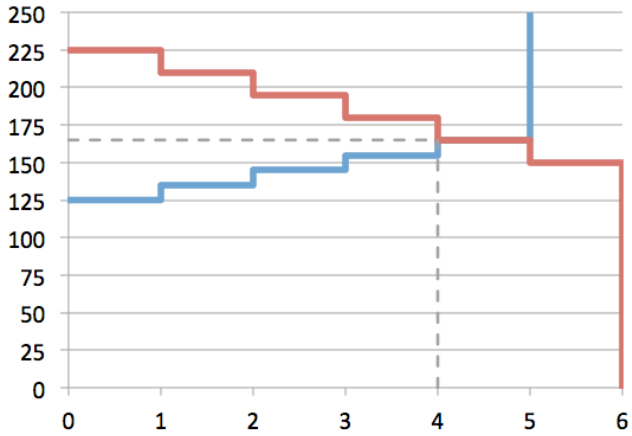

Figure 4.2: adding the supply curve for five traders (in blue). The equilibrium point is given by the intersection of the supply and demand curves: the equilibrium price is $1.655 (indicated by the horizontal dashed line) and the equilibrium quantity is 4 (vertical dashed line).

In BSE, the parameters in `order_schedule` determine the minimum and maximum prices on the supply and demand curves, and the nature of how the values between the minimum and maximum are created (i.e., via steps that are either `fixed`, `random`, or `jittered`), but the number of steps on each curve is determined by how many traders there are working sell orders, and how many there are working buy orders – that is, by the number of sellers and the number of buyers in the market. These two numbers are specified separately, in the `traders_spec` parameter that is discussed in more detail in Section 5, below.

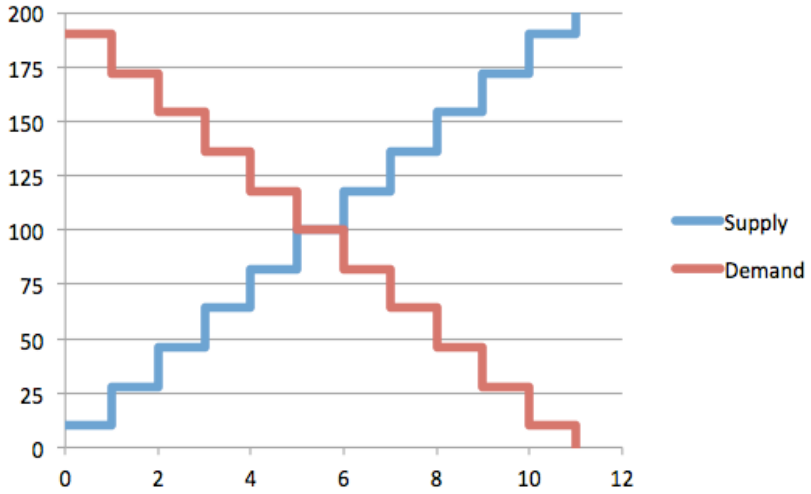

Figure 4.3: supply and demand curves when both `supply_schedule` and `demand_schedule` are set by `'ranges':[(10,190)]` and `'stepmode':'fixed'`.

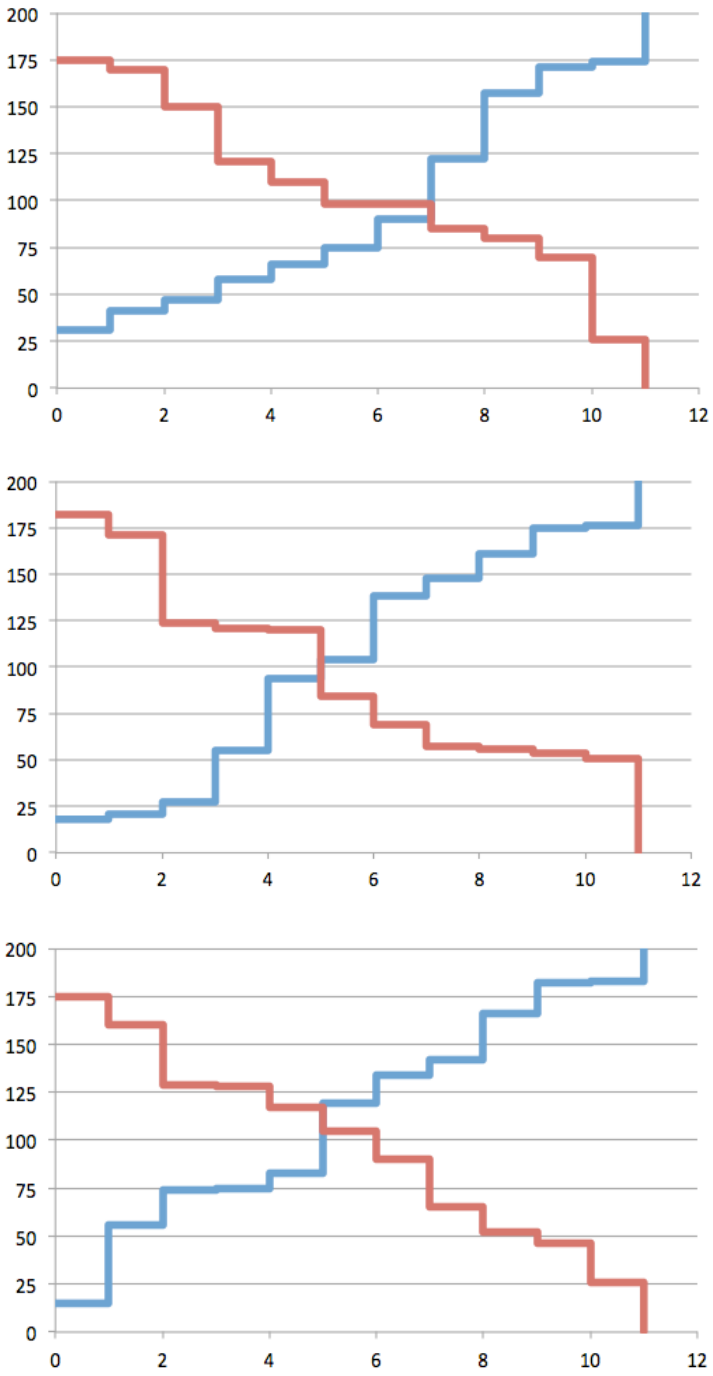

Figure 4.4: three example supply and demand curves generated when both schedules are set by `'ranges':[(10,190)]` and `'stepmode':'random'`.

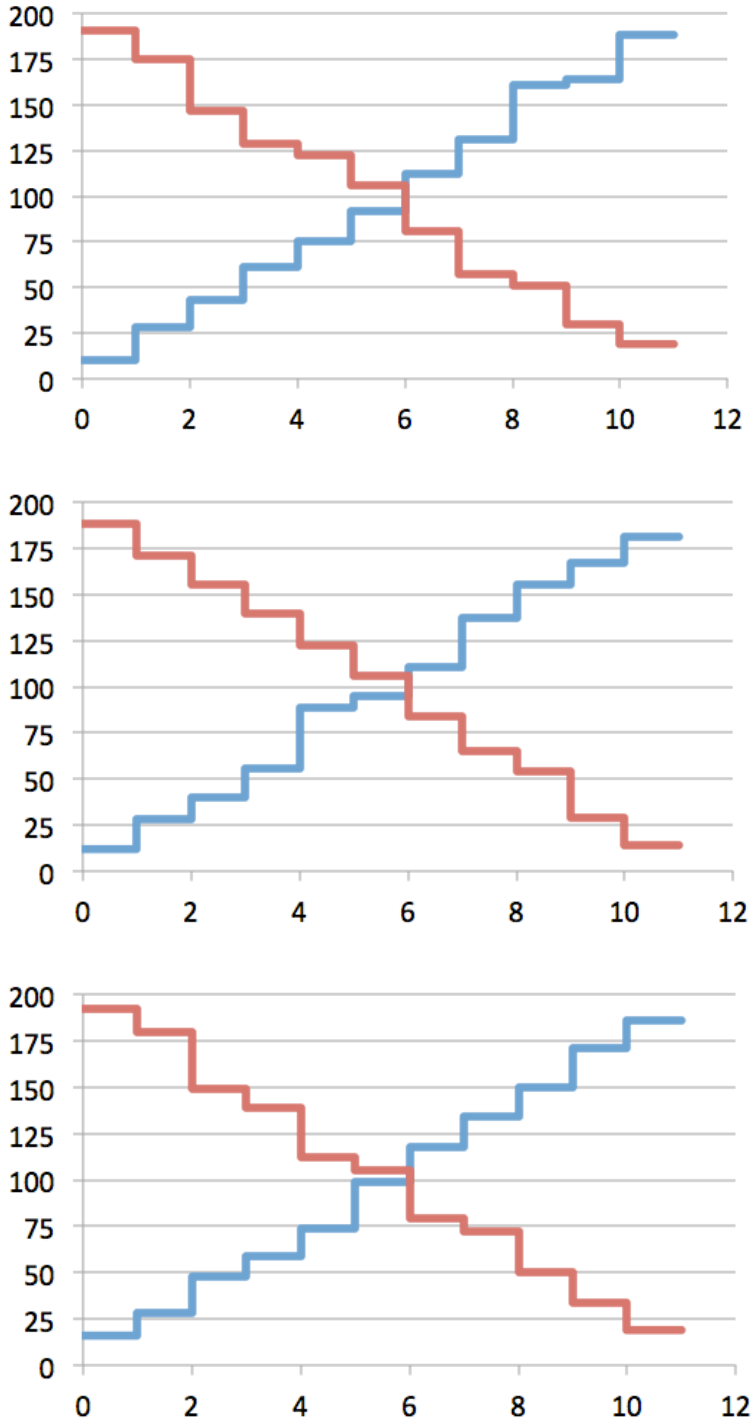

Figure 4.5: three example supply and demand curves generated when both schedules are set by `'ranges':[(10,190)]` and `'stepmode':'jitter'`.

If `stepmode` is set to `'random'`, then more than one range tuple can be specified in `ranges`, in which case `customer_orders()` then generates each order price by first choosing one of the range tuples at random (with each tuple being given equal probability) and then generating a number at random within that tuple. An example of the kind of supply and demand curve that this can be used to create is shown in Figure

4.6. Note that it is only when `stepmode` is `'random'` that any additional range tuples in `ranges` are used: if `stepmode` is set to `'fixed'` or `'jittered'` and `ranges` contains more than one tuple, only the first tuple is used.

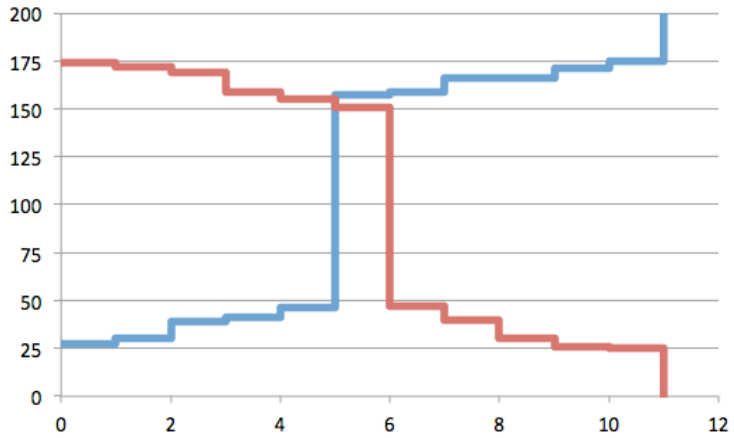

Figure 4.6: example supply and demand curves generated when both the supply and the demand schedules are set by `'ranges':[(25,50),(150,175)]` and `'stepmode':'random'`.

So far, we've looked at how `customer_orders()` allows you to specify a single pair of supply and demand curves which are used for the duration of an experiment. However, in real markets the supply and demand curves change over time, meaning that the equilibrium price in the market is also very likely to dynamically alter. In the current version of `customer_orders()`, it is possible to switch between different supply and demand schedules over the duration of a market session. So, each individual supply or demand curve is specified as a Python dictionary data-structure, as we have seen previously, and the set of schedules is then a list of such dictionaries, as illustrated in the following code snippet:

```
range1 = (10,190)
range2 = (200,300)

sup_sched = [ {'from':0, 'to':60, 'ranges':[range1], 'stepmode':'fixed'},
              {'from':60, 'to':120, 'ranges':[range2], 'stepmode':'fixed'},
              {'from':120, 'to':180, 'ranges':[range1], 'stepmode':'fixed'}
            ]
dem_sched = sup_sched
```

Code Snippet 4.2: setting up demand and supply schedules that each hold constant for one minute, switching from Range1 to Range2 and then back to Range1 over three minutes. The equilibrium price of the market when supply and demand are both set by range1 is $1.00; the equilibrium price of the market when supply and demand are both set by range2 is $2.50.

Snippet 4.2 defines a schedule that runs for three minutes, where for the first minute the supply and demand are as illustrated in Figure 4.3, with an equilibrium price of $1.00; then for the second minute the supply and demand curves are both shifted up to the range [$2.00, $3.00] with an equilibrium price of $2.50; and then in the final third of the experiment the supply and demand curves are re-set to those that were used in the first minute, so the equilibrium price reverts to $1.00. Having the equilibrium price vary over the course of the market session is one way of increasing the realism of the experiment: in the snippet above, we've introduced a couple of "shock changes" where the equilibrium jumps suddenly to a new value. It's important to ensure that automated traders can react appropriately to such shocks, and also to continuous smooth changes in the equilibrium price.

Another way in which realism can be increased is to have the customer orders arrive in a continuous random stream, rather than periodically having every single trader being given a new customer order, all at the same instant. Such full periodic replenishment of customer orders is something that was introduced in Vernon Smith's first ever experiments (reported in Smith, 1962) and which very many experimenters have used since, but there are good reasons for wanting to explore simulation experiments where instead the replenishment is continuous: in very many real-world markets, for much of the time, the flow of orders is much more like a continuous random drip-feed of orders into the market, rather than long periods of zero new orders punctuated by regular brief intense bursts when large numbers of new orders arrive. For more discussion of this, see Cliff & Preist (2001), De Luca *et al.* (2011), and Cartlidge & Cliff (2012).

In `customer_orders()`, the time-distribution of new customer orders is specified via the `timemode` of the schedule. In the example output shown in Section 3 we had set `timemode` to `'periodic'`. We can alter this to a continuous drip-feed by instead setting `timemode` to one of the following three values:

- `'drip-fixed'`: for a schedule in which n traders are each to be re-supplied with orders over a period of p seconds, individual orders arrive regularly at a fixed interval of p/n seconds between each order, so the arrival time of the i^{th} order is $t=i*(p/(n-1))$ for $n>1$.

- `'drip-jitter'`: similar to `'drip-fixed'`, but the arrival time of the i^{th} order is given by $t=i*(p/n)+\mathbf{U}(0, p/n)$ where $\mathbf{U}(x, y)$ generates a random value from a uniform continuous distribution in the range $[x, y]$. Thus the longest possible inter-arrival time is $2p/n$.

- `'drip-poisson'`: here the arrival of new orders is modelled as a *poisson process*, and hence the inter-arrival times of orders follows an exponential distribution.

The random nature of the inter-arrival times when `timemode` is `'drip-jitter'` or `'drip-poisson'` means that generating a sequence of n new orders may result in the final order arrival time being at a time other than p; if this happens, it is detected by `customer_orders()` and the arrival times are then scaled up or down so that the n^{th} order arrives at time p: this means that all three of the continuous-replenishment `timemodes` can be thought of as pseudo-periodic: the supply or demand schedule will

fully replenish every *p* seconds. This pseudo-periodicity is useful for maintaining appropriate control over supply and demand when running comparative experiments, but it can be disabled if required (by setting the internal Boolean flag `fittointerval` to `False`).

So, if we use the schedules shown in Snippet 4.2, resupplying every 30 seconds and with `timemode` set to `'drip-fixed'`, the kind of transaction-price time-series that we see are shown in Figure 4.7.

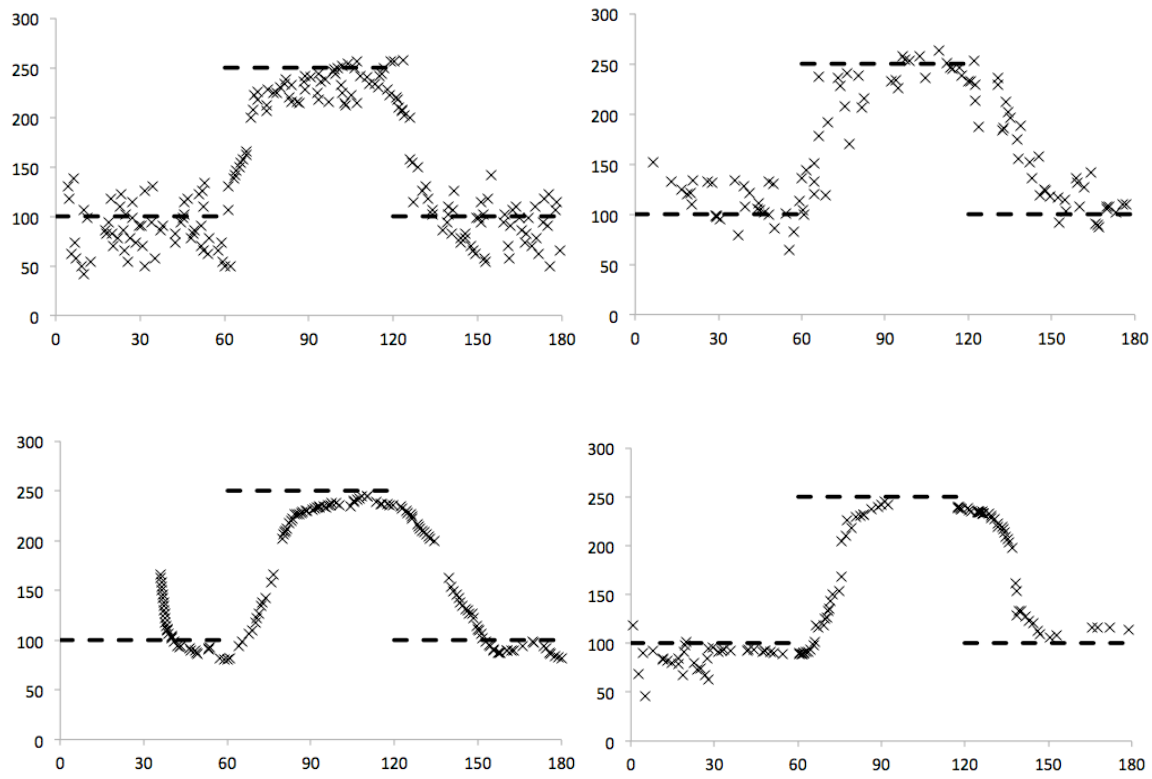

Figure 4.7: example transaction-price time-series from markets whose supply and demand curves are set by the code in Snippet 4.2, with continuous replenishment of orders. The equilibrium price P_0 is indicated by the horizontal dashed line: in the first minute P_0=\$1.00, then at t=60 there is a "shock change" to P_0=\$2.50, and then after a further minute the equilibrium price undergoes another shock change at t=120 back to P_0=\$1.00. In each time-series, the response to the step-change in the equilibrium price as the schedule moves from `range1` to `range2` at t=60 and back from `range2` to `range1` at t=120 is clear. In each market there are 40 buyers and 40 sellers: graph at top left shows results from market populated by Giveaway traders; top right shows results from ZIC traders; bottom left shows results from Shaver traders; and bottom right shows results from ZIC traders.

Switching between different supply and demand schedules, to introduce "shock changes" in the market, is common practice in experimental economics and studies of automated trading systems. Although a market with shock changes is manifestly more challenging than one in which the supply and demand curves remain unchanged

throughout the experiment, it could still be argued that such experimental markets differ from many real-world markets in one important regard: apart from the occasional step-change when the "shocks" occur, the supply and demand (and hence the equilibrium point) remain constant for long periods. In many real markets of major significance, transaction prices do not settle to some fixed value and hold at that equilibrium in a flat line; instead, the transaction prices are constantly varying, and those variations in transaction prices probably reflect constant dynamic changes in the underlying equilibrium point of the market, as the supply curve and demand curve dynamically vary.

In the current version of BSE, it is possible to introduce dynamic variation of the supply and demand curves by providing additional parameters in the schedule's range. As before, the first two parameters in the range are the minimum and the maximum value son the schedule; but if a third parameter is supplied, that is interpreted as an offset function `offset(t)`, which specifies an offset value, that is added to the schedule's minimum and maximum values at time *t*. If the supply schedule and the demand schedule have the same offset function, then the supply curve and demand curve are shifted up or down equally by the offset function at each time *t* and hence the equilibrium price is altered. This is best illustrated by an example: code snippet 4.3 shows an offset function that is a sine wave that grows in amplitude, and reduces its wavelength, as time progresses. The baseline supply and demand curves in this example are each flat, a constant value of $0.95 for all sellers and a constant $1.05 for all buyers (experimental economists refer to this as a "box" schedule), but the variations in the offset function will move the curves off the baseline as time progresses.

```
# schedule_offsetfn returns time-dependent offset on schedule prices
def schedule_offsetfn(t):
    pi2 = math.pi * 2
    c = math.pi*3000
    wavelength = t/c
    gradient = 100*t/(c/pi2)
    amplitude = 100*t/(c/pi2)
    offset = gradient + amplitude * math.sin(wavelength*t)
    return int(round(offset,0))

rangeS = (95, 95, schedule_offsetfn)
rangeD = (105, 105, schedule_offsetfn)

sup_sched = [ {'from':0, 'to':600, 'ranges':[rangeS], 'stepmode':'fixed'} ]

dem_sched = [ {'from':0, 'to':600, 'ranges':[rangeD], 'stepmode':'fixed'} ]
```

Code Snippet 4.3: setting up demand and supply schedules that are each a single constant value for all buyers and all sellers, but which are then shifted up and down over the 10 minutes of the experiment by a sine wave that grows in amplitude over time, and decreases in wavelength over time, as specified in the offset function.

Figure 4.8 shows a transaction-price time series from one experiment run using the schedules set up in Snippet 4.3.

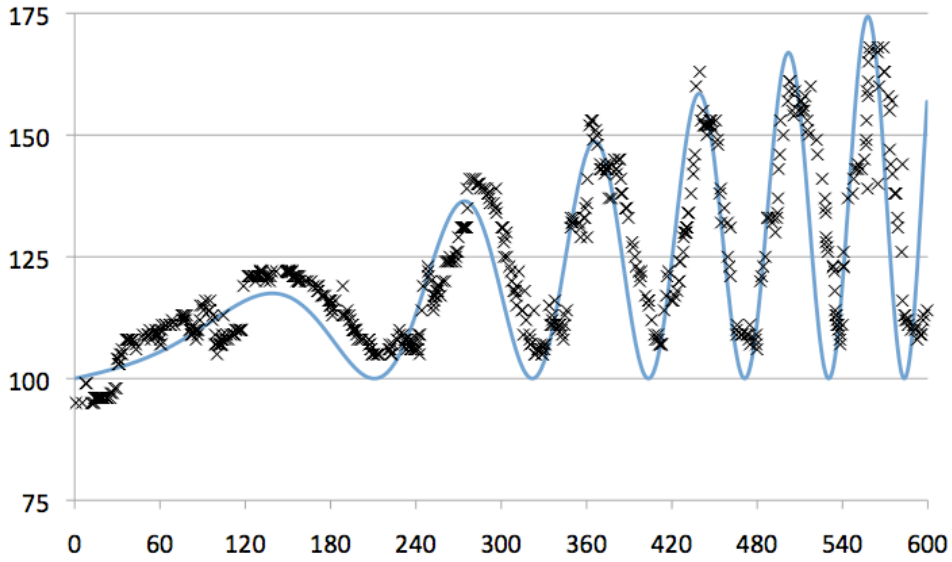

Figure 4.8: transaction price time-series from a market where the supply and demand schedules in Snippet 4.3 are used, with 40 buyers and 40 sellers, each group divided into 10 Giveaway traders, 10 Shavers, 10 ZIC and 10 ZIP. The sinusoidal solid blue line indicates the market's equilibrium price, the midpoint of supply and demand schedules at each time *t*; the crosses show individual transaction prices.

If only one offset function is specified in a schedule range, that offset function is applied equally to all prices in the schedule. For more sophisticated dynamic variation of supply and demand, it is possible to specify two offset functions in the schedule range, in which case the first function (third value in the tuple) is the time-varying function applied to the range minimum, and the second function (fourth value in the tuple) is the time-varying function applied to the range maximum. So, if we were to define four separate offset functions, the code in Snippet 4.3 could be altered to read as follows:

```
rangeS = (95, 95, S_sched_min_offsetfn, S_sched_max_offsetfn)
rangeD = (105, 105, D_sched_min_offsetfn, D_sched_max_offsetfn)
```

Note that, in real financial markets, the dynamic variations in prices are often modeled mathematically as a "drunkard's walk", the kind of time series that you'd get if, at each time-step, you add a small random value to whatever the price was on the previous time-step. Approximations to this kind of time-series can be made in BSE by generating random values in the offset function: for example, in Snippet 4.3, random values from specific distributions could be added to `gradient`, `amplitude` and `wavelength` on each call.

5. Comparing different robot traders

Figures 5.1 and 5.2 show the transaction-price time series, and the time series of accumulated profit, resulting from a single call to `market_session()` with 20 Giveaway traders, 20 Shavers, 20 ZICs, and 20 ZIPs, each split equally into 10 buyers and 10 sellers, specified in Snippet 5.1, using the supply and demand schedules that had previously been specified in Snippet 4.2.:

```
order_sched = {'sup':sup_sched, 'dem':dem_sched,
               'interval':30, 'timemode':'drip-poisson'}

buyers_spec = [('GVWY',10),('SHVR',10),('ZIC',10),('ZIP',10)]
sellers_spec = buyers_spec
traders_spec = {'sellers':sellers_spec, 'buyers':buyers_spec}
```

Code Snippet 5.1: setting up for an experiment comparing four types of trader.

The market session was then run for three simulated minutes (180 seconds) starting at t=0, using the sequence of instructions in Snippet 5.2:

```
dumpfile=open('test1data.csv','w')
market_session('test1', 0, 180, traders_spec, order_schedule, dumpfile, True)
tdump.close()
```

Code Snippet 5.2: running the experiment.

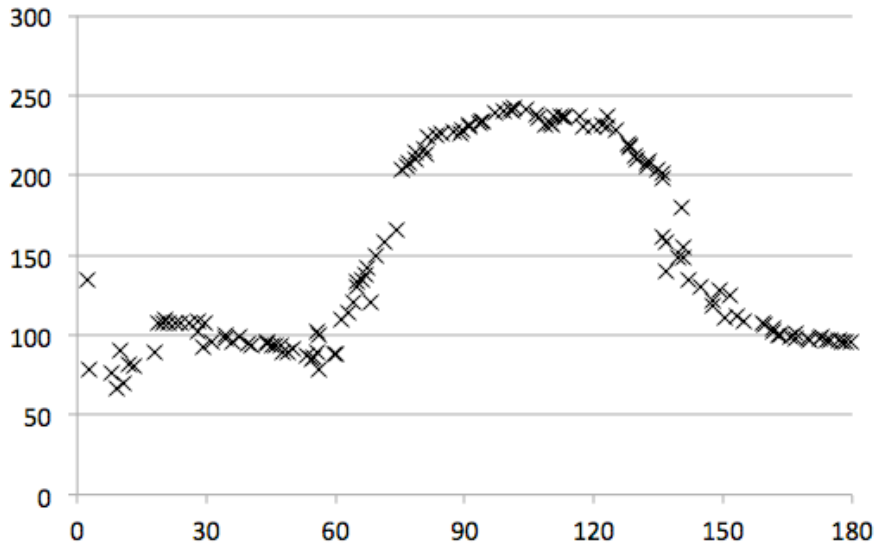

Figure 5.1: transaction-price time-series from three-way comparative test among 20 Giveaway, 20 Shaver, 20 ZIC, and 20 ZIP traders. Horizontal axis is time; vertical axis is price.

The final parameter to `market_session()` is the Boolean flag `dump_each_trade` which determines how much output is written to the data dump-file: if `True`, then a line of summary data is written after every trade in the market session; if `False` then all that is written is a line of summary data for the situation at the end of the session.

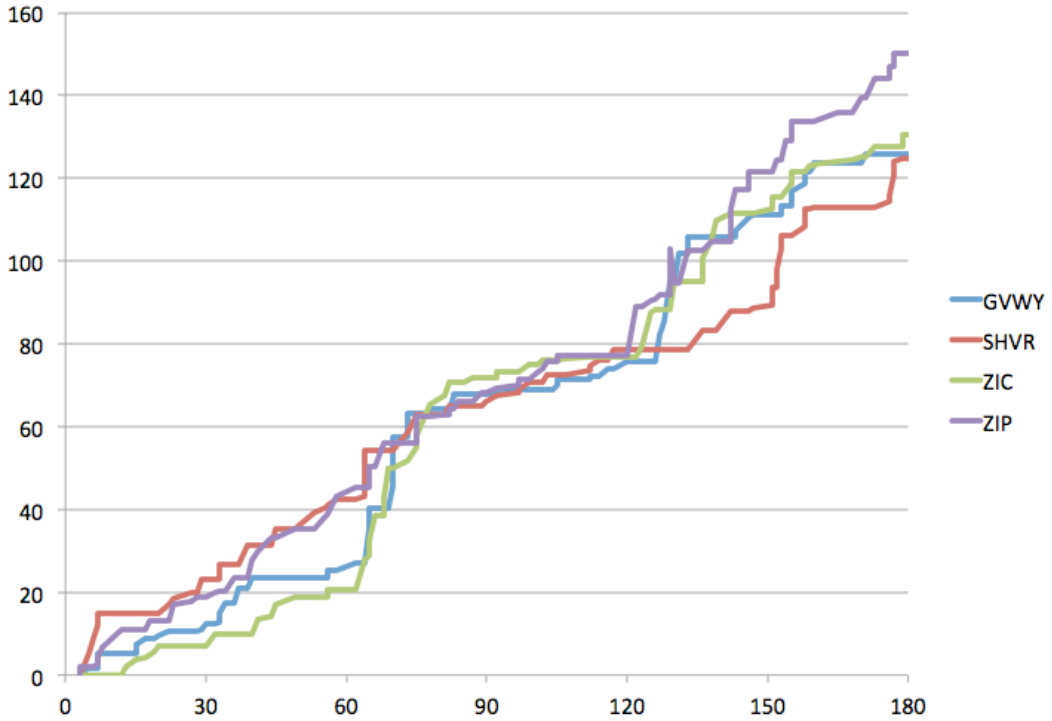

Figure 5.2: Accumulated average profit per trader-type in the four-way comparison experiment whose transaction-price time-series was illustrated in Figure 5.1.

So, at the end of the session, as illustrated in Figure 5.2, the average accumulated profit per trader is best for ZIPs and lowest for Shavers. But this is just a single session, and the random assignment of orders to traders will have had some effect: maybe ZIP just got lucky.

To run a proper comparative study, we need to do multiple sequences of statistically independent sessions, and then calculate appropriate summary statistics from, and/or perform appropriate tests of statistical significance on, the results. This is easily done, as shown in Snippet 5.3:

```
tdump=open('bigtestdata.csv','w')
trial = 1
while (trial<1000):
    tid = 'trial%04d' % trial
    market_session(tid, 0.0, 180, traders_spec, order_schedule, tdump, False)
    trial = trial + 1
tdump.close()
```

Code Snippet 5.3: running an experiment of 1000 independent trials.

The file `bigtestdata.csv` that Snippet 5.3 produces will have 1000 lines in it, each line being the average accumulated profit per trader, for the four types of robot, after a session of 180 simulated seconds of market activity – in each of the 1000 trails, the sequence of order allocations is random, so the same sequence is unlikely to occur twice. The data from the 1000 sessions can then be analyzed to determine which type of robot makes the most profit on average in this particular test, with this specific pair of supply and demand schedules, and this particular 20:20:20:20 ratio of robot types. Altering the supply and demand or the ratio of robot types might change things significantly: establishing which robot trader performs best across a number of different supply and demand schedules, and with differing numbers and ratios of robot types, can require an awful lot of trials and tests.

Snippet 5.4 shows code for systematically varying the ratios of trader-types in a market, keeping the ratio identical on the supply side and the demand side for each experiment. For four types of trader, and with an equal-balance ratio of 4:4:4:4 giving 16 buyers and 16 sellers, and with 50 trials for any one ratio, the code generates a sequence of 22,750 trials, and takes several hours to complete on a single laptop computer. For more comprehensive evaluations, run-times of days or weeks of continuous CPU time may be required, at which point it is better to split the run across multiple machines, perhaps via Amazon Web Services.

Designing an appropriately structured set of experiments, and analyzing the potentially very large amounts of data that they generate, is a topic explored in depth by books such as those by Antony (2003), Cohen (1995), Field & Hole (2003), Janert (2010), and Montgomery (2012).

Footnote: How realistic is all this?

The rise of the Web enabled people outside of the finance industry to readily access sources of market data over the internet, and to connect directly to brokerages, and so there has been a rise of interest in "day trading", where professional or semi-professional independent traders work from home or from rented offices, connecting to market-data sources and buying and selling shares and other tradable instruments via their broker. A quick trawl of the Web reveals a variety of stock-market simulators, many of them aimed at day-traders wanting to evaluate an automated strategy. Such simulators often work with historical data of stock prices: by back-testing on historical data, it is possible to estimate how much money an automated trading strategy would have made (or lost) if it had been running live.

Clearly, BSE is quite different to such trading simulators, but there are good reasons for that. Very few trading simulators work with Level 2 data (i.e. with the full LOB, changing order-by-order), and the cost of obtaining such data is often very high. More fundamentally, trading simulators based on historical market data typically cannot model *market impact*, where buying or selling large amounts of an instrument shifts the demand and/or supply curves in such a way that the equilibrium price of the instrument then alters. In that sense, conventional trading simulators require the trader (human or automated) to act solely as a *price-taker*, trading at time *t* at whatever price is on the screen, whatever price the historical data says the instrument was trading at, at time *t.* Whether they sell 1 share or 10million shares at time *t*, the price immediately after the

sale will still be whatever the historical data says it was – yet a sale of 10million shares would in reality almost definitely move the price down, in a way that selling 1 share simply wouldn't. Trading robots in BSE can be *price-makers*, and their activity can shift the supply and demand, and they can generate, and have to deal with, market impact. In that sense, the scenario in BSE is more like a modern-day "dark pool", a private online exchange where a relatively small number of traders meet to conduct big transactions (participants in dark pools are typically traders working for major banks or fund-management companies, dealing large blocks of tradable instruments).

De Luca *et al.* (2011) and Cartlidge & Cliff (2012) discuss the need to explore trading agents in simulated markets that are more realistic than those used in prior experimental studies of robot traders interacting with one another and/or with human traders. The current version of BSE offers can be configured to run "traditional" experiments switching between static-equilibrium supply/demand schedules and with periodic simultaneous replenishment of orders, but it can also be configured to have continuous "drip-feed" replenishment, and fine-grained dynamic variations in the supply and demand schedules, and hence also in the market equilibrium.

Nevertheless, in comparison to real markets, the lack of any latency in the system is the probably the biggest issue. It would be relatively easy to introduce simulated latency at the exchange (so the LOB data that is received by the traders at time t actually reflects the state of the LOB a little earlier, at time $t-\Delta t$) and we could also introduce "communications latency" so that when a trader issues an order at time t it does not arrive at the exchange until a little later at some time $t+\Delta t$, but in a single-threaded simulation it would require quite a lot more work to accurately model processing latency in each trader. That is, in the current version of BSE, each trader gets as long as it wants to process its `respond()` method, whereas in a reality a lot of effort goes into making the response-time of automated trading systems as low as possible while still being capable of generating profitable behaviours. To better model real market systems, we would need to switch to a multi-threaded simulation, and/or to configuring BSE as a distributed client-server architecture over multiple virtual or physical machines. That remains for further work.

```
n_trader_types = 4
equal_ratio_n = 4
n_trials_per_ratio = 50
n_traders = n_trader_types * equal_ratio_n

fname = 'balances_%03d.csv' % equal_ratio_n

tdump=open(fname,'w')

min_n = 1

trialnumber = 1
trdr_1_n = min_n
while trdr_1_n <= n_traders:
      trdr_2_n = min_n
      while trdr_2_n <= n_traders - trdr_1_n:
            trdr_3_n = min_n
            while trdr_3_n <= n_traders - (trdr_1_n + trdr_2_n):
                  trdr_4_n = n_traders - (trdr_1_n + trdr_2_n + trdr_3_n)
                  if trdr_4_n >= min_n:
                        buyers_spec = [('GVWY', trdr_1_n),
                                       ('SHVR', trdr_2_n),
                                       ('ZIC',  trdr_3_n),
                                       ('ZIP',  trdr_4_n)]
                        sellers_spec = buyers_spec
                        traders_spec = {'sellers':sellers_spec,
                                        'buyers':buyers_spec}
                        print buyers_spec
                        trial = 1
                        while trial <= n_trials_per_ratio:
                              trial_id = 'trial%07d' % trialnumber
                              market_session(trial_id, start_time, end_time,
                                             traders_spec, order_sched, tdump,
                                             False)
                              tdump.flush()
                              trial = trial + 1
                              trialnumber = trialnumber + 1
                  trdr_3_n += 1
            trdr_2_n += 1
      trdr_1_n += 1
tdump.close()

print trialnumber-1
```

Code Snippet 5.4: running an experiment with 32 traders (16 buyers and 16 sellers) of types Giveaway, Shaver, ZIC, and ZIP; with the ratio of the four types of traders being systematically varied across all possible nonzero values, and performing 50 independent trials for each specific ratio. This requires a total of 22,750 trials to be performed, and on a single CPU would take several hours of continuous computation.

References and Further Reading

J. Antony (2003), *Design of Experiments for Engineers and Scientists.* Butterworth.

J. Cartlidge & D. Cliff (2012), "Exploring the 'robot phase transition' in experimental human-algorithmic markets" UK Government Office for Science, Foresight Driver Review DR25, http://www.bis.gov.uk/assets/foresight/docs/computer-trading/12-1058-dr25-robot-phase-transition-in-experimental-human-algorithmic-markets.pdf.

D. Cliff (1997), "Minimal Intelligence Agents for Bargaining Behaviours in Market-Based Environments". Hewlett-Packard Labs Technical Reports HPL-97-91, http://www.hpl.hp.com/techreports/97/HPL-97-91.pdf.

D. Cliff & C. Preist (2001), "Days without end: on the stability of experimental single-period continuous double auction markets". Hewlett-Packard Labs Technical Reports HPL-2001-325. http://www.hpl.hp.com/techreports/2001/HPL-2001-325.pdf.

P. Cohen (1995), *Empirical Methods for Artificial Intelligence.* MIT Press Bradford Books.

R. Das, J. Hanson, J. Kephart, & G. Tesauro (2001), "Agent-human interactions in the continuous double auction" *Proceedings of the International Joint Conference on Artificial Intelligence* (IJCAI'01), http://spider.sci.brooklyn.cuny.edu/~parsons/courses/840-spring-2005/notes/das.pdf

M. De Luca, J. Cartlidge, C. Szostek, & D. Cliff (2012), "Studies of interactions between human traders and algorithmic trading systems" UK Government Office for Science, Foresight Driver Review DR13, http://www.bis.gov.uk/assets/foresight/docs/computer-trading/11-1232-dr13-studies-of-interactions-between-human-traders-and-algorithmic-trading-systems.pdf.

A. Field & G. Hole (2003), *How to Design and Report Experiments.* Sage.

S. Gjerstad & J. Dickhaut (1998), "Price formation in double auctions", *Games and Economic Behavior,* **22**(1):1-29. http://pdf.aminer.org/000/164/313/price_formation_in_double_auctions.pdf.

D. Gode & S. Sunder (1993), "Allocative efficiency of markets with zero-intelligence traders: Market as a partial substitute for individual rationality", *Journal of Political Economy*, **101**(1):119-137. http://www2.econ.iastate.edu/tesfatsi/Gode%20and%20Sunder-JPE.pdf

P. Janert (2010), *Data Analysis with Open Source Tools*. O'Reilly

D. Montgomery (2012), *Design and Analysis of Experiments*, 8th Edition. John Wiley.

J. Rust, J. Miller, & R. Palmer (1992), "Behaviour of trading automata in a computerized double auction market" in D. Friedman & J. Rust (editors) *The Double Auction Market: Institutions, Theories, & Evidence.* Addison Wesley, pp.155-198. http://time.dufe.edu.cn/jingjiwencong/waiwenziliao/behtradingdat.pdf.

V. Smith (1962), "An experimental study of competitive market behavior" *Journal of Political Economy*, **70**(2):111-137. http://www.cob.ohio-state.edu/~young_53/Smith.pdf.

P. Vytelingum, (2006), *The Structure and Behaviour of the Continuous Double Auction*. PhD Thesis, University of Southampton, School of Electronics and Computer Science. http://eprints.soton.ac.uk/263234/.

Programming Exercise

NB: BSE was written for use in undergraduate and masters-level teaching in computer science at the University of Bristol. Here is an example programming assignment that was set as a piece of assessed coursework, for the first release of BSE in October 2012.

As a programming exercise, implement one or more additional types of trader-robot in BSE, explore the behaviour of the new traders by running sensibly structured comparative experiments, and then analyze the results from those experiments using appropriate visualization and statistical analysis methods. Write a brief report that clearly explains how the additional trader robots work, the design and analysis of the experiments, and their outcome.

You might want to implement someone else's trading algorithm, such as "GD" by Gjerstad & Dickhaut (1998), which, like ZIP, was demonstrated in modified ("MGD") form by Das *et al.* (2001) to outperform human traders; or you might want to try the more recent "AA" by Vytelingum (2006), which currently seems to be the best published trading strategy for market experiments like these; see De Luca *et al.* (2011) and Cartlidge & Cliff (2012) for further discussion.

But you may also want to experiment with writing your own algorithm. It is fine if you want to start by implementing a previously-published algorithm like ZIP or MGD or AA, and then alter or extend it – that's how a lot of progress in science and engineering is made. But if you want to start from scratch, that is absolutely fine too.