COMSM0048 - Internet Economics and Financial Technology - Coursework Option 2020

Guide for creating a minimal DIMM01 (Dumb Illustrative Market Maker) in BSE.

This document is a step-by-step guide for creating a DIMM01 "prop" trader in the Bristol Stock Exchange.

https://github.com/davecliff/BristolStockExchange/blob/master/BSE.py

In its current form, the experiment code in BSE.py (the stuff right at the end of the source file) is set up to treat the various automated traders in BSE as "sales traders" designated as either buyers or sellers. That's because the (entirely artificial) division of traders into buyer and seller roles for the duration of an experiment has been used very frequently in experimental economics, both with humans and with trading agents. So, the current experiment code echoes that tradition, but if you look at the actual BSE implementation of the trading agents such as ZIP, they are not hard coded as either buyers or sellers, they can take either role.

From the perspective of adding one or more market-makers to BSE, the existence of a liquid market of sales traders, of buyers and sellers all engaged in trading, is a good thing: the more other traders there are generating activity in the market, the better for a market-maker (MM) agent, because there should be more opportunities for the MM to trade.

So, it seems sensible to leave the BSE.py experiment code largely as it is, and just add in the necessary code for dealing with the introduction of one or more MM agents.

Let's start by adding a new class of trader, a market maker -- a **D**umb, **I**llustrative, **M**arket **M**aker (DIMM). As we may want to play with more than one type of DIMM, let's call the first one DIMM01. For the time being, lets create DIMM01 as a straight clone of GVWY – this is absolutely not a market-maker, but at least it is a trader agent that we know works. We can come back and turn it into a proper MM later. For the time being, we just want something called a DIMM01, so we can alter the experiment code to integrate the DIMM01 into everything that's already there. So, here is GVWY cloned as DIMM:

Algorithm 1: create `Trader_DIMM01` class

```
class Trader_DIMM01(Trader):

        # the following is just a copy of GVWY's getorder method
        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,lob['QID'])
                return order
```

(NB: code in green ink is duplicated, cut and pasted in from the `GVWY` definition of `getorder()`; code in blue ink is novel, freshly-written for the DIMM01 class.).

Now let's go right down to the endzone of the BSE.py file, to the bit where the experiments are set up and run. Setting up for an experiment previously involved creating a `buyers_spec` and a `sellers_spec` and then packaging those two up into a single object, a `traders_spec`. But now we need to extend the `traders_spec` so it includes a specification of one or more marketmakers, which we'll refer to as the `mktmakers_spec`. Let's just add one DIMM01 trader as a marketmaker:

Algorithm 2: add market makers to the traders' specification

```
buyers_spec = [('GVWY',10),('SHVR',10),('ZIC',10),('ZIP',10)]
sellers_spec = buyers_spec
mktmakers_spec = [('DIMM01',1)]
traders_spec = {'sellers':sellers_spec, 'buyers':buyers_spec, 'mktmakers':mktmakers_spec}
```

(Note: old code is in green ink, new code is in blue).

The `traders_spec` is passed into `market_session()`, and the first substantive thing that `market_session()` does is a call to `populate_market()`, which constructs the population of traders from the data in `traders_spec`. As we've just extended `traders_spec` to include specification of market makers, so clearly `populate_market()` will need some extending too.

When we look in the definition of `populate_market()` there is a local `trader_type()` method which uses an `if-elif-elif-…else` statement reading the `robottype` (the short codename for the robot type, such as 'GVWY" or 'SHVR', used in the `traders_spec`) to decide which class of trader to create and instantiate – clearly this needs extending to include DIMM01. So, the new trader-type should be something like this:

Algorithm 3: Defining `Trader_DIMM01` in `populate_market()`

```
def trader_type(robottype, name):
        if robottype == 'GVWY':
            return Trader_Giveaway('GVWY', name, 0.00, 0)
        elif robottype == 'ZIC':
            return Trader_ZIC('ZIC', name, 0.00, 0)
        elif robottype == 'SHVR':
            return Trader_Shaver('SHVR', name, 0.00, 0)
        elif robottype == 'SNPR':
            return Trader_Sniper('SNPR', name, 0.00, 0)
        elif robottype == 'ZIP':
            return Trader_ZIP('ZIP', name, 0.00, 0)
        elif robottype == 'DIMM01':
            return Trader_DIMM01('DIMM01', name, 500.00, 0)
        else:
            sys.exit('FATAL: don\'t know robot type %s\n' % robottype)
```

Note that the constructor for DIMM01 is passed an opening balance of 500, rather than zero. This is because, unlike the sales traders which can make a profit via executing client orders, for MM agents they are trading on their own account, and so they need to go into the market with an endowment of some kind: either some money to buy with or some stock to sell or both. In this simple example we're just giving our one DIMM trader a starting balance of $500. If we had specified more than one DIMM trader in the `mktmakers_spec`, the way this code is written would give all DIMM traders the same opening balance (you may want to change that if you decide to experiment with more than one MM in your market).

The main code for the `populate_market()` method as currently written has a loop where the number of buyers in the spec is counted, and each buyer is given a unique ID; immediately followed by cut-and-paste-similar code where the number of sellers in the spec is counted, and each seller is given a unique ID. We can do another cut-and-paste job to add in the necessary code for marketmakers, as illustrated below (in blue):

Algorithm 4: `populate_market()` edits to include market makers (insert blue code)

```
n_buyers = 0
for bs in traders_spec['buyers']:
        ttype = bs[0]
        for b in range(bs[1]):
                tname = 'B%02d' % n_buyers # buyer i.d. string
                traders[tname] = trader_type(ttype, tname)
                n_buyers = n_buyers + 1

if n_buyers < 1:
        sys.exit('FATAL: no buyers specified\n')

if shuffle: shuffle_traders('B', n_buyers, traders)

n_sellers = 0
for ss in traders_spec['sellers']:
        ttype = ss[0]
        for s in range(ss[1]):
                tname = 'S%02d' % n_sellers # seller i.d. string
                traders[tname] = trader_type(ttype, tname)
                n_sellers = n_sellers + 1

if n_sellers < 1:
        sys.exit('FATAL: no sellers specified\n')

if shuffle: shuffle_traders('S', n_sellers, traders)

n_mktmakers= 0
for ms in traders_spec['mktmakers']:
        ttype = ms[0]
        for m in range(ms[1]):
                tname = 'M%02d' % n_mktmakers # mktmaker i.d. string
                traders[tname] = trader_type(ttype, tname)
                n_mktmakers = n_mktmakers + 1

if n_mktmakers < 1:
        sys.exit('FATAL: no marketmakers specified\n')

if shuffle: shuffle_traders('M', n_mktmakers, traders)

if verbose :
        for t in range(n_buyers):
                bname = 'B%02d' % t
                print(traders[bname])
        for t in range(n_sellers):
                bname = 'S%02d' % t
                print(traders[bname])
        for t in range(n_mktmakers):
                bname = 'M%02d' % t
                print(traders[bname])

return {'n_buyers':n_buyers, 'n_sellers':n_sellers, 'n_mktmakers':n_mktmakers}
```

So now when we populate the market, we've added in the specified number of market-maker agents. Remember, the call to `populate_market()` came at the start of the `market_session()` method. The next action, after that, within `market_session()` is to compute the `timestep`, the nominal fraction of a second that passes between each trader being processed, such that all traders can be processed in one second. To keep it consistent, that needs an obvious minor extension:

Algorithm 5: Edit the `timestep` (blue code) within `market_session()`

```
# timestep set so that can process all traders in one second
# NB minimum interarrival time of customer orders may be much less than this!!
timestep =
1.0/float(trader_stats['n_buyers']+trader_stats['n_sellers']+trader_stats['n_mktmakers'])
```

And now, if you run the code after making the edits listed above, it should all work: a single DIMM01 trader is created, along with all the buyers and all the sellers that we had before. And, as currently configured, the DIMM01 trader then simply sits there and does nothing: remember, DIMM01's internal mechanism at the moment is just GVWY, which just takes a client order and quotes at a give-away price. But because DIMM01 is not a sales trader it is never given any client orders to execute, so it never buys and it never sells. It just hangs around with its money in the bank, oblivious of what's going on in the market.

Nevertheless, with these few minor edits and extensions, DIMM01 is now integrated into the BSE experiments. We can now go back to the DIMM01 class definition and start to turn it into something that really does trade on its own account, buying and selling to make a profit. What we build here in this example will be spectacularly dumb, but it will be illustrative of the basics; and at least now we know that as we extend and test it, it will be interacting with the other traders, the buyer and seller sales-traders, in the market.

The simplest sort of prop trader or market maker that I can think of in the context of BSE would be one that implements a minimal "long only" (i.e., buy-low, hold, and then sell-high) strategy with a maximum holding of one unit. The following pseudocode implements this "long only" strategy:

Algorithm 6: Pseudocode for a DIMM01 "long only" strategy

```
1.1    if (I am not holding a unit)
1.2        and (someone is offering a unit for sale)
1.3        and (I have enough money in my bank to pay the asking price)
1.4    then
1.5        (buy the unit)
1.6        (remember the purchase-price I paid for it)
1.7    if (I am holding a unit)
1.8    then
1.9       (my asking-price is that unit's purchase-price plus my profit margin)
1.10      if (there is a buyer bidding more than my asking price)
1.11      then
1.12         (sell my unit to the buyer and put the money in my bank)
```

The idea being that the pseudocode in lines 1.1 to 1.12 is embedded in some kind of repeat-forever loop: the MM trader is perpetually either seeking to buy a unit using the money it holds or seeking to sell the unit it holds for a profit to add to its bank account. Such is the life of a prop trader.

There are very many ways in which we could criticize this strategy, and very many ways in which we could improve it. But let's stick with this for the time being and concentrate on how we get it into the definition of DIMM01 within BSE.

We can note here that lines 1.1-1.5 are basically about *buying something* from the market; line 1.6 is a bit of book-keeping that we need to take care of so that we remember what we've paid; and then lines 1.7-1.12 are basically about *selling something* back into the market. In principle the Python implementation of lines 1.1-1.5 could be as complex as a ZIP or MGD or AA buyer-agent (or more complex than that even); and similarly, the Python implementation of lines 1.7-1.12 could be as complex as a seller running ZIP, MGD, AA, or anything more complex (or indeed any combination of these kind of strategies). But we're aiming for a minimally simple example here, so we'll stick to implementing the pseudocode as directly as possible.

Now if you look at the definition of ZIP in BSE.py, you can see that it overloads the `respond()` method that is declared as a stub in the `Trader` superclass, implementing a method for allowing ZIP traders to watch what's happening on the LOB and responding by updating their internal state accordingly. We can do the same in DIMM. First of all, let's express the pseudocode lines 1.1-1.5 in proper Python:

Algorithm 7: `respond()` method of `Trader_DIMM01`

```
def respond(self, time, lob, trade, verbose):
	# DIMM buys and holds, sells as soon as it can make a "decent" profit
	if self.job == 'Buy':
		# see what's on the LOB
		if lob['asks']['n'] > 0:
			# there is at least one ask on the LOB
			bestask = lob['asks']['best']
			# try to buy a single unit at price of bestask+biddelta
			bidprice = bestask + self.bid_delta
			if bidprice < self.balance :
				# can afford it!
				# do this by issuing order to self, processed in getorder()
				order=Order(self.tid, 'Bid', bidprice, 1, time, lob['QID'])
				self.orders=[order]
				if verbose : print('DIMM01 Buy order=%s ' % ( order))
```

…and then continuing in the same method we can deal with lines 1.7-1.12 too:

Algorithm 8: `respond()` method of `Trader_DIMM01` continued

```
	elif self.job == 'Sell':
		# is there at least one counterparty on the LOB?
		if lob['bids']['n'] > 0:
			# there is at least one bid on the LOB
			bestbid = lob['bids']['best']
			# sell single unit at price of purchaseprice+askdelta
			askprice = self.last_purchase_price + self.ask_delta
			if askprice < bestbid :
				# seems we have a buyer
				# do this by issuing order to self, processed in getorder()
				order=Order(self.tid, 'Ask', askprice, 1, time, lob['QID'])
				self.orders=[order]
				if verbose : print('DIMM01 Sell order=%s ' % ( order))
	else :
		sys.exit('FATAL: DIMM01 doesn\'t know self.job type %s\n' % self.job)
```

Thus far, all seems well, but we still have some work to do. In writing this version of `respond()` we have implied that the DIMM trader has internal variable like `self.bid_delta` and `self.last_purchase_price` that we will need to make sure are declared and initialized in the DIMM's version of `__init__()`, the object constructor. Also, we have not addressed the book-keeping needed for line 1.6 of the pseudocode. And we need to make sure that the orders that a DIMM agent issues to itself in this version of `respond()` actually get executed appropriately (and that it does the right thing if the orders happen not to be executed).

(Now it's written out in Python, it should be clear just how naïve this DIMM01 strategy really is: there is no sense of being influenced by past events or of estimating the likelihood of future expectations. DIMM01 buys when it can afford to buy, and it sells as soon as it can find a willing counterparty who will buy at a price which gives DIMM01 the minimum profit that it seeks in a trade. Don't bet your life savings on the performance of this trader. Anyhow, back to the coding…)

Let's deal with the easy stuff first. To create and instantiate the new variables local to `Trader_DIMM01`, we need to give that class its own ***local*** `__init__()` constructor, like this:

Algorithm 9: `__init__()` constructor of `Trader_DIMM01`

```
    def __init__(self, ttype, tid, balance, time):

        super().__init__(ttype, tid, balance, time) # initialise Trader superclass

        # below are new variables for the DIMM only (they are not in the superclass)
        self.job = 'Buy' # flag switches between 'Buy' & 'Sell' to show what DIMM does next
        self.last_purchase_price = None # we haven't traded yet so set null
        self.bid_delta = 1 # how much (absolute value) to improve on best ask when buying
        self.ask_delta = 5 # how much (absolute value) to improve on purchase price
```

NB: all of this is new code for the new `__init__()` constructor of the `Trader_DIMM01` class.

Now the last thing to do is to take care of the book-keeping. For a prop trader or a market maker, money is made by selling things at a price higher than that paid for them. So, unlike a sales trader, a prop trader bears the whole cost of purchasing an item and, again unlike a sales trader, a prop trader receives the whole of the income from the sale of an item. We also need to remember that when a DIMM trader buys an item it should then switch to selling that item, and when a DIMM trader sells an item it should then switch to bidding to buy a new item. We can do all that by creating a `bookkeep()` method ***local*** to DIMM, ***overloading*** the one that would otherwise be inherited from the `Trader` superclass (note: do not edit the `bookkeep()` method in the `Trader` superclass - we can leave that one alone as we don't want to mess with the book-keeping of all the other traders ; we only want to introduce new logic specific to the `Trader_DIMM01` class), as follows:

Algorithm 10: `Trader_DIMM01`'s local book-keeping method.

```
def bookkeep(self, trade, order, verbose, time):

    outstr = ""
    for order in self.orders:
        outstr = outstr + str(order)

    self.blotter.append(trade) # add trade record to trader's blotter
    # NB What follows is **LAZY** -- it assumes all orders are quantity=1
    transactionprice = trade['price']

    bidTrans = True #did I buy? (for output logging only)
    if self.orders[0].otype == 'Bid':
        # Bid order succeeded, remember the price and adjust the balance
        self.balance -= transactionprice
        self.last_purchase_price = transactionprice
        self.job = 'Sell' # now try to sell it for a profit
    elif self.orders[0].otype == 'Ask':
        bidTrans = False # we made a sale (for output logging only)
        # Sold! put the money in the bank
        self.balance += transactionprice
        self.last_purchase_price = 0
        self.job = 'Buy' # now go back and buy another one
    else:
        sys.exit('FATAL: DIMM01 doesn\'t know .otype %s\n' %
                 self.orders[0].otype)

    self.n_trades += 1

    verbose = True # We will log to output

    if verbose: # The following is for logging output to terminal
        if bidTrans: # We bought some shares
            outcome = "Bght"
            owned = 1
        else:        # We sold some shares
            outcome = "Sold"
            owned = 0
        net_worth = self.balance + self.last_purchase_price
        print('%s, %s=%d; Qty=%d; Balance=%d, NetWorth=%d' %
              (outstr, outcome, transactionprice, owned, self.balance, net_worth))

    self.del_order(order) # delete the order
```

Notice here that we're treating any unit held by the trader as contributing to that trader's "net worth", and assuming that its value has not changed since it was purchased. This is a fairly major simplifying assumption: the market price for selling the asset may have changed since it was purchased. (Economists have spent a lot of time looking at markets in which the value of an asset decays over time: rural Spanish fish markets in particular where a lack of air-conditioning and refrigeration mean that a fish fresh landed at sunrise commands a high price yet is worth very much less at sunset.)

Now, having written `response()` and made the specified edits to the local versions of `__init__()` and `bookkeep()`, we can keep the GVWY code for `getorder()` because the order-prices that DIMM01 sets include the margin priced in at the point the order is generated (this, again, is a simplification).

And that's it, that's all we need to get a dumb, illustrative market-maker up and running.

In the code you have downloaded, all the X_`verbose` flags are set to `False`. You can find them in the `market_session` method, as below (where `X = {orders, lob, process, respond, bookkeep, populate}`):

Algorithm 11: verbose flags for logging are set to `False`. Set to `True` to see output to console.

```
        # one session in the market
        def market_session(sess_id, starttime, endtime, trader_spec,
                           order_schedule, tdump, dump_all, verbose):

                orders_verbose = False
                lob_verbose = False
                process_verbose = False
                respond_verbose = False
                bookkeep_verbose = False
                populate_verbose = False
```

Adjust the "verbose" flags to `True` to see logging output to terminal. This will help you to understand and debug code.

The code you have downloaded is also set to run `n_trials=6` sessions, with fixed `range1` and `range2` values for the supply and demand schedules, as shown below:

Algorithm 12: Code is set to run 6 trials, with a fixed range demand and supply schedules.

```
 # The code below sets up symmetric supply and demand curves
    # at prices from 50 to 150, P0=100

    range1 = (50, 150)
    supply_schedule = [{'from': start_time, 'to': end_time, 'ranges': [range1],
                       'stepmode': 'fixed'}]

    range2 = (50, 150)
    demand_schedule = [{'from': start_time, 'to': end_time, 'ranges': [range2],
                       'stepmode': 'fixed'}]

...

    # n_trials is how many trials (i.e. market sessions) to run in total
    n_trials = 6
```

When you run your BSE code without making edits to the code in Algorithms 11 and 12, the console will only show each time a new session starts and each time DIMM01 executes a trade. You can easily change what is logged to console by setting the verbose flags to `True` (Algorithm 11).

In the example output in Figure 1, below, `populate_verbose=True` so we can see each trader that is initialized. Also, this run used supply and demand schedules where there was a growing sinusoidal oscillation in the price, but with the market steadily rising. This was achieved by editing the code in Algorithm 12 to:

```
range1 = (50, 150, schedule_offsetfn)
range2 = (50, 150, schedule_offsetfn)
```

As you can see, the "NetWorth" value for the DIMM01 trader increases over the course of this run. A simple strategy such as DIMM01 should be expected to make some money in a market that is rising overall, so the fact that this console output does show DIMM01 making money is really just an indication that nothing is seriously broken in the code: in almost any other kind of market, DIMM01 is unlikely to make much money at all, but that is because it is DIMM. Your job is to design and implement something cleverer. Good luck!

Figure 1: Terminal output with populate_verbose logging

```
[TID B00 type GVWY balance 0.0 blotter [] orders [] n_trades 0 profitpertime 0]
[TID B01 type SHVR balance 0.0 blotter [] orders [] n_trades 0 profitpertime 0]
[TID B02 type SHVR balance 0.0 blotter [] orders [] n_trades 0 profitpertime 0]
...
[TID S38 type ZIC balance 0.0 blotter [] orders [] n_trades 0 profitpertime 0]
[TID S39 type ZIC balance 0.0 blotter [] orders [] n_trades 0 profitpertime 0]
[TID M00 type DIMM01 balance 500.0 blotter [] orders [] n_trades 0 profitpertime 0]

sess0001;

[M00 Bid P=051 Q=1 T= 1.99 QID:5], Bght=50; Qty=1; Balance=450, NetWorth=500
[M00 Ask P=055 Q=1 T= 3.93 QID:15], Sold=67; Qty=0; Balance=517, NetWorth=517
[M00 Bid P=143 Q=1 T= 4.10 QID:17], Bght=142; Qty=1; Balance=375, NetWorth=517
[M00 Ask P=147 Q=1 T=447.80 QID:18620], Sold=149; Qty=0; Balance=524, NetWorth=524
[M00 Bid P=160 Q=1 T=447.94 QID:18626], Bght=159; Qty=1; Balance=365, NetWorth=524
[M00 Ask P=164 Q=1 T=515.04 QID:21393], Sold=164; Qty=0; Balance=529, NetWorth=529
[M00 Bid P=158 Q=1 T=518.83 QID:21551], Bght=157; Qty=1; Balance=372, NetWorth=529
[M00 Ask P=162 Q=1 T=575.20 QID:23939], Sold=165; Qty=0; Balance=537, NetWorth=537
[M00 Bid P=168 Q=1 T=575.42 QID:23949], Bght=167; Qty=1; Balance=370, NetWorth=537
```

If `populate_verbose = True` then each trader is output to console upon initialisation. We see the sales' traders begin with `balance=0.0` while `DIMM01` begins with `balance=500.00`. Session 1 begins and the `bookkeep` method (Algorithm 10) in `Trader_DIMM01` outputs an updated account for the market maker each time it trades. At time T=1.99, the market maker M00 enters a bid at price 51, which executes at price 50. The market maker now holds one share and has a balance of 450. At time 3.93, the market maker enters an ask for price 55, which executes at price 67. The market maker M00 now holds no shares and has a balance of 517 (i.e., it has made a profit of 17). In this market, the `DIMM01` trader ends up with a total net worth of 537.

[END]