Agent Based Model

Introduction

I've always been interested in Agent Based Models (ABM). An ABM works by simulating a population of agents, who can interact with each other and with the environment in programmed ways, in order to see how they behave as an aggregate population. This is often using, for example, in macroeconomics, where an ABM might take the following form: assume we have a group of firms, a group of customers, and a government. The firms will interact with the customers both by selling them goods and by employing them, and with the government through the payment of taxes. The customers also pay taxes to the government. And the government might have control over monetary policy such as interest rates.

And in this defined "world", the various agents might interact, and we can observe what happens and extend those results to estimate what might happen in the real world.

In my opinion, the actual construction of ABMs are under-documented on the internet, and so I sought out to build a simple one of my own to get started. Here I will simulate a marketplace, which will have traders (sellers) and shoppers (buyers). Each trader will have a list of goods that they sell, and each shopper will have a shopping list of goods that they need. In this ABM, physical space matters - the traders have fixed positions within the "world", and the shoppers start out at random locations and must spend time to travel to each trader stall, looking for the goods they need.

And so a potential motivation for designing such a model might be to ask: If this world represented an amusement park, how could we think about the optimal layout of the stalls that would optimize sales? Or maybe minimize customer traveling time? Now, we won't completely answer those questions with this basic model that I'm getting started with, but this is an example of how I would think about the problem.

I'll also do this in base R code, so that no packages are required. This code will be a little difficult at times to break up into coherent "chunks" for piecemeal discussion, and so the entire code is listed at the very end.

Constants Definition

First we will declare a few costants that will help us set up the "world" of our simulation. First we choose the number of paths. Since many of our processes will be randomized, we don't just want to run the simulation once - we want to run it many times and aggregate the results, in a Monte Carlo-like fashion. And so here we will conduct 100 completely separate runs of the simulation. Each of those simulations we will allow to run for a total of 200 arbitrary time units.

In the next few lines after that, we define some features of the agents in the model. Each of the traders will stock 5 items (these will be randomly selected later). There will be 10 shoppers in the model, and each shopper will have a walking speed of 0.5 arbitrary units. Then we define the physical dimensions of our simulation space as a 30x30 sqaure (again in arbitrary units):

```
rm(list=ls()) # clear all variables
set.seed(1126) # set seed for reproducibility

# SECITON: USER INPUTS
num_paths = 100 # the number of runs of the simulation (think Monte Carlo paths)
max_time = 200  # the max # of time steps before simulation ends

n_items_stocked = 5 # the number of items each trader will carry
num_shoppers = 10 # the number of shoppers in the simulation
```

```
walking_speed = 0.5 # some unitless constant for the walking speed of each shopper
width_x = width_y = 30 # the unitless dimensions of our simulation space

# initialize a storage object to hold results from each of the num_paths:
master_past_storage = array( NA , c( max_time+1 , num_paths ) )
```

Setup

We continue by defining some more specific features of the simulation.

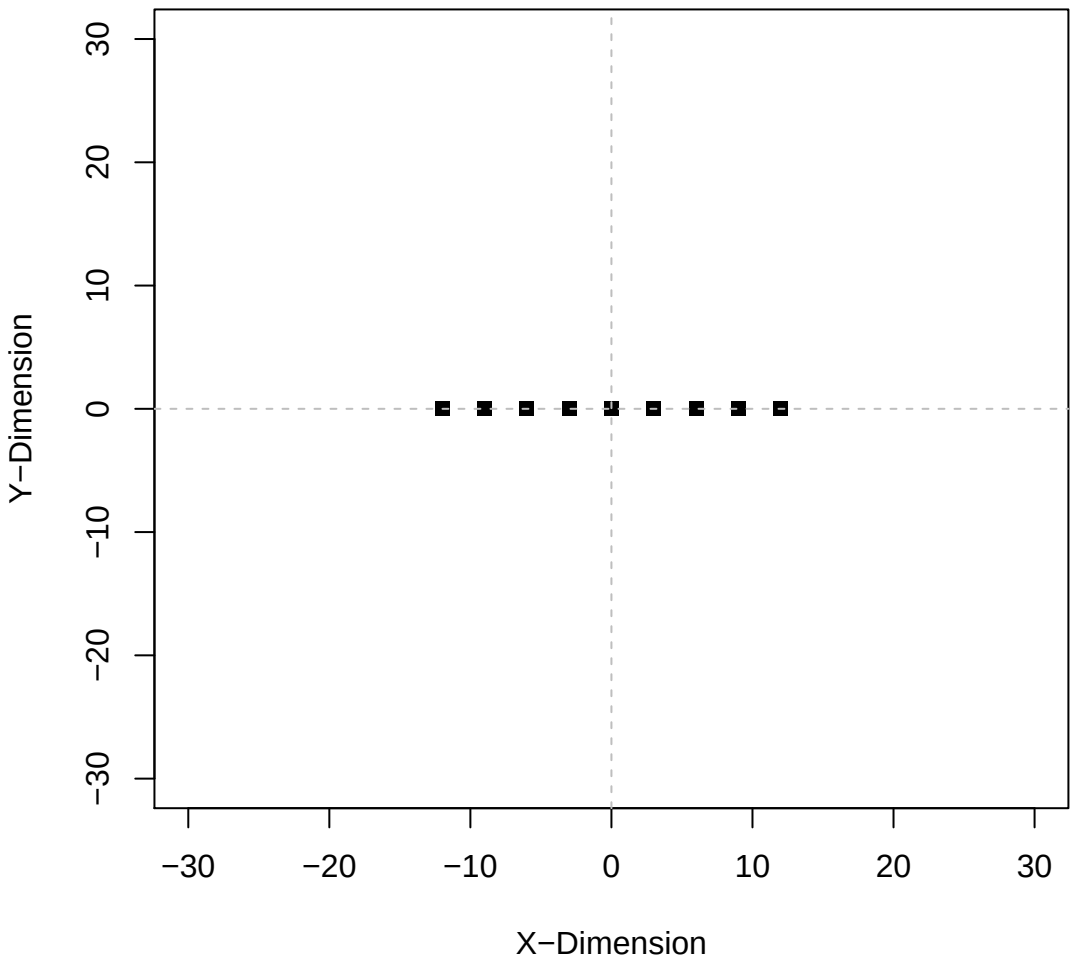
In the block of code below, the first thing we do is define the "universe" of shoppable items. We manually define a list of 12 items that are able to be bought by shoppers and sold by traders, using fruits and vegetables as examples:

```
fruit_and_veg = c(
  "apples", "bananas","oranges","plums","mangoes","grapes",
  "cabbage","potatoes","carrots","lettuce","tomatoes","beans"  )
```

Now we have to give more definition to our traders and their stalls. Our grid is a 30x30 space, but as an initial simplifying assumption we will only allow trader stalls to be positioned along the y=0 axis (so, as we look down on the space we can picture a horizontal line of stalls from left to right). So all we need to do is define the x coordinates of those stalls, and the number of coordinates we give defines the number of stalls:

```
# positions of a row of stalls
xs = c( -12, -9, -6, -3, 0, 3, 6, 9, 12 )


plot( x = xs ,
      y = rep(0, length(xs) ),
      pch = 15 ,
      xlim = c(-width_x,width_x),
      ylim = c(-width_y,width_y),
      xlab = 'X-Dimension',
      ylab='Y-Dimension')
abline( h = 0 , lwd = 1, lty = 2 , col = 'gray' )
abline( v = 0 , lwd = 1 , lty = 2 , col = 'gray' )
```



EXECUTION

Now things start getting a little harder to explain in a linear text format, because we'll be doing a lot of looping. Our outermost loop, for example, will be to go through each of the paths we defined at the very start:

```
for( path in 1:num_paths ) {
```

```
  time = 0  # initialize the time steps to 0

  # initialize arrays for storage of trader inventories,
  # shopper coordinates, and shopper lists
  trader_stock = array( NA, c( length(xs) , n_items_stocked ) ,
                        dimnames = list( xs,NULL) )
  shopper_coords = array( NA , c(num_shoppers, 2 ) ,
```

```
                          dimnames = list( NULL, c('x','y')))
shopper_list = array( NA , c(num_shoppers, 8 ) )
closest_stall_name = array( NA , c(num_shoppers))
stalls_visited_memory = array( 0 , c( num_shoppers , length(xs) ) )
```

What we've done here is first initialized our time counter and set it to 0. Then we've initialized some "storage" variables that will come in handy later as we generate and track different data points throughout the simulation. In the future, this is one section that can probably be cleaned up: some of these separate tracking arrays can be consolidated into a series of lists that would make it neater.

Next, for each trader stall we'll determine their inventory by randomly sampling from the possible choices:

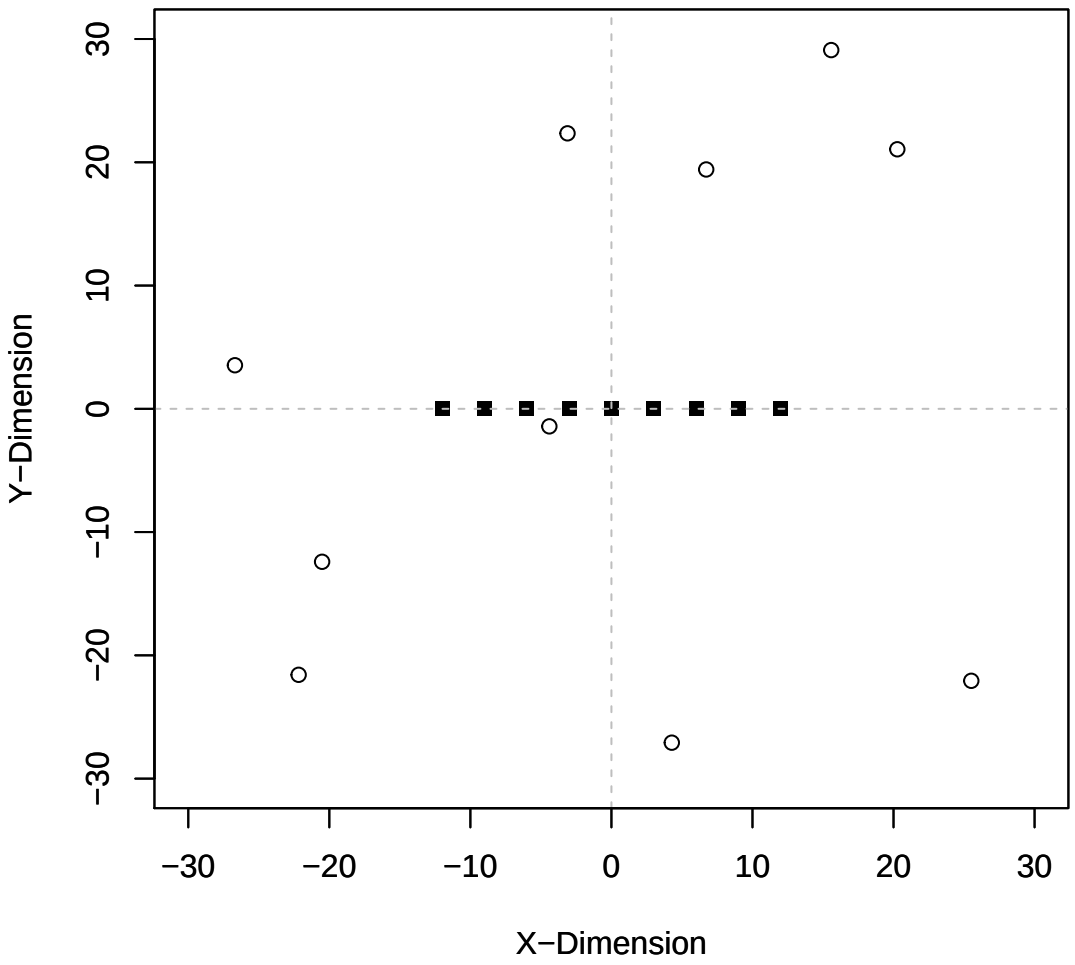
```
for( i in 1:length(xs) ) {
  trader_stock[i,] = sample( x = fruit_and_veg,
                             size = n_items_stocked,
                             replace = FALSE )
}
```

Similar to the manner in which we just set up our traders, now we will set up our shoppes. For each of the total number of shoppers we selected, we'll generated random x and y coordinates for them. Then we'll determine what's on their shopping list. One interesting thing to note here is that although we defined every trader stall to carry 5 and exactly five items, here we allow the shoppers to have shopping lists of variable size, between 1 and 8 items. So some shoppers might only need 1 item and then they are done, whereas others might need up to 8 items before being done:

```
for( j in 1:num_shoppers) {
  shopper_coords[j, 1 ] = runif( n = 1, min = -width_x, max = width_x)
  shopper_coords[j, 2 ] = runif( n = 1, min = -width_y, max = width_y )
  num_items = sample( 1:8 , 1 )
  shopper_list[j,(1:num_items)] = sample( x = fruit_and_veg,
                                          size = num_items,
                                          replace = FALSE )
}
```

Let's take a look at what our simulation space looks like now, with both the trader stalls and the shoppers added:

```
plot( x = xs ,
      y = rep(0, length(xs) ) ,
      pch = 15 , xlim = c(-width_x,width_x),
      ylim = c(-width_y,width_y),
      xlab = 'X-Dimension',
      ylab='Y-Dimension')
abline( h = 0 , lwd = 1, lty = 2 , col = 'gray' )
abline( v = 0 , lwd = 1 , lty = 2 , col = 'gray' )
par(new=TRUE)
plot( x = shopper_coords[,1],
      y = shopper_coords[,2],
      pch = 1,
      xlab = "X-Dimension",
      ylab = "Y-Dimension",
      xlim = c(-width_x,width_x),
      ylim = c(-width_y,width_y) )
```



And now we get to start looking at things we'd like to measure. A key metric we'd like to track is the average number of items remaining on the shopping lists of all shoppers. This number is useful because when it drops to zero it means that all the shoppers have finished. So if our optimization goal was to minimize the time it takes for shoppers to find what they need, we would want to get this number to zero as quickly as possible.

But for now we just initialize a variable to track it:

```
# compute the mean number of items remaining among all shoppers
num_items_remaining = apply( shopper_list , 1 , FUN = function(x){
  length(na.omit( x ) )    })
num_items_remaining_avg = mean(num_items_remaining)
storage_num_items_remaining = num_items_remaining_avg
```

SIMULATION BEGINS

First we start a while loop, which will continue based on the maximum amount of simulation time that we defined earlier. Although we could also set this up if we wanted to continue until the number of items left to buy was zero:

```
while( ( time < max_time ) ) {
```

Then, for each moment in time, we will loop through all shoppers:

```
# loop through all shoppers:
for( shopper in 1:num_shoppers ) {
```

For now let's just set this equal to the first shopper for illustration:

```
shopper=1
```

The first thing we do for each shopper is check to see if they still have anything left to buy. We do this becuse a shopper with nothing left on their list can effectively be ignored in our model - they have no further actions to take:

```
# check if the selected shopper has something left to buy?
if(
  (length( na.omit( shopper_list[ shopper , ] ) ) > 0 ) &
  any( stalls_visited_memory[shopper,]==0)
) {
```

The next chunk does a few things. First, we define a custom function to compute the distance from a given single shopper to all available trader stalls.

Next, it checks to see if the currently selected shopper has already pointed themselves toward a stall. Initially the answer to this will be "no" for all shoppers, because they have not made any decisions yet.

After that, we use the custom function to compute those distances between shopper and stalls. We then use this vector of distances to choose the closest stall, but we also do this *after* removing any stalls that the shopper has already visited. This gives each shopper a "memory" of the stalls they have visited before:

```
SHOPPER_TO_STALL_DISTANCE <- function(
  shopper_xy = shopper_coords[1,] ,
  stall_x = 0,
  stall_y = 0
) {
  dist = sqrt(  ( stall_x - shopper_coords[shopper,1] )^2 +
                  ( stall_y - shopper_coords[shopper,2] ) ^2  )
  return( dist )
}
```

```
# check if the shopper has already selected a stall to head towards?
# if not, have them select the closest stall
if( is.na( closest_stall_name[shopper] ) ) {

  # compute distance from the current shopper to all possible stalls
  stall_distances =
```

```
    sapply( X = xs , FUN = function(x){
      SHOPPER_TO_STALL_DISTANCE( shopper_xy = shopper_coords[shopper,],
                                 stall_x = x,
                                 stall_y = 0)   })
  names(stall_distances) <- xs
  # compare that against this shopper's "memory" and
  # remove those that have already been visited
  stall_distances =
    stall_distances[ which( stalls_visited_memory[ shopper , ] == 0 ) ]
  if( length(stall_distances) != 0 ){
    closest_stall_name[shopper] = as.numeric(names( which.min( stall_distances )))
  }
  closest_stall_index = which( xs == closest_stall_name[shopper] )
}

# compute distance to the stall already selected
selected_stall_distance =
  SHOPPER_TO_STALL_DISTANCE( shopper_xy = shopper_coords[shopper,],
                             stall_x = xs[closest_stall_index],
                             stall_y = 0 )
```

Now comes another important piece of the code.

At each time step, besides picking the stall they should head towards, each shopper faces another decision rule. If they are close enough to their selected stall (where we define "close enough" as being within 0.25 of our arbitrary distance units), then they will conduct business at that stall, purchasing all of the items on their list that are stocked by that stall.

Note a few assumptions here: on the one hand, we simplify things by currently assuming that each trader stall has an unlimited inventory of each of the items it stocks. So although they only carry certain items, they will never sell out of an item that they carry.

Second, in this version of the model we ignore pricing - the traders do not charge a given price and the shoppers are similarly assumed to be price agnostic or to have unlimited funds (if we had a model "to-do" list, fixing this would probably be the very next iteration of this model).

However, one interesting feature we incorporate is that a shopper does not "know" the inventory of a stall until they are within the shopping range. So it's very possible for a shopper to spend time approaching a stall that does not in fact carry anything that they need. This lack of perfect information that the shoppers face is important to incorporate since the goal of our model is to ultimately optimize marketplace efficiency.

If the shopper is *not* in range of their selected stall, then the other portion of the code block below computes the mechanics of their two-dimensional travel across the simulation space, bringing them closer to their selected stall. A shopper cannot move and shop in the same time step:

```
# check if the shopper is close enough to the selected stall,
# if yes, have them conduct shopping,
# if no then have the shopper move toward the selected stall:
if ( selected_stall_distance <= 0.25 ) {  # <--if close enough, then purchase

  # check the stall inventory against the shopper's required list
  inventory = trader_stock[ closest_stall_index , ]
  shop_list = na.omit( shopper_list[shopper,] )
  matched_list = shop_list[ shop_list %in% inventory ]
  remaining_list = setdiff( shop_list , inventory )
  shopper_list[shopper,] = NA
```

```
  if( length(remaining_list) != 0 ) {
    shopper_list[shopper,1:length(remaining_list)] = remaining_list
  }

  # and then also "remember" that this shopper has
  # already been to this stall
  stalls_visited_memory[ shopper , closest_stall_index  ] = 1
  # now reset the selected trader stall,
  # so that a new one is selected on the next go-around
  closest_stall_name[shopper] = NA

}  else  {  #  <-- if NOT close, then move the shopper

  angle = atan2(
    y = ( 0 -  shopper_coords[shopper,2]  ) ,
    x = (  xs[closest_stall_index] - shopper_coords[shopper,1]  )  ) # *(360)
  shopper_coords[shopper,1] = shopper_coords[shopper,1] +
    ( walking_speed * cos(angle) )
  shopper_coords[shopper,2] = shopper_coords[shopper,2] +
    ( walking_speed * sin(angle) )

}
```

And that concludes the current logic for shoppers and their behavior through time. To be thorough, we would at this point then close the IF statement that checked the shopper's shopping list length, and then close the FOR loop for that customer:

```
    }  # <-- this one closes loop that checks customer shopping list length

}  # <-- this one closes the customer loop
```

So now we would return to the above, and repeat everything for shopper #2.

Once we were done computing the decisions for all shoppers in this current time step, we would again compute our performance metric for the average number of items remaining, and add that to our tracking array, so that we can track this value through time:

```
# compute the mean number of items remaining among all shoppers
num_items_remaining = apply( shopper_list , 1 , FUN = function(x){
  length(na.omit( x ) )    })
num_items_remaining_avg = mean(num_items_remaining)
storage_num_items_remaining =
  append( storage_num_items_remaining ,  num_items_remaining_avg )
```

Then we would increment our time variable, close the time loop, and then repeat the above process for all shoppers at the next point in time. Once we had iterated through all of our time points, we would now have the results of a single complete simulation.

In the current form, these results would take the form of a single time series: the average number of shopping list items remaining at each point in time. We would save that single time series to a master array that we initialized earlier, where each column in this master array represents one of our 100 simulation paths:

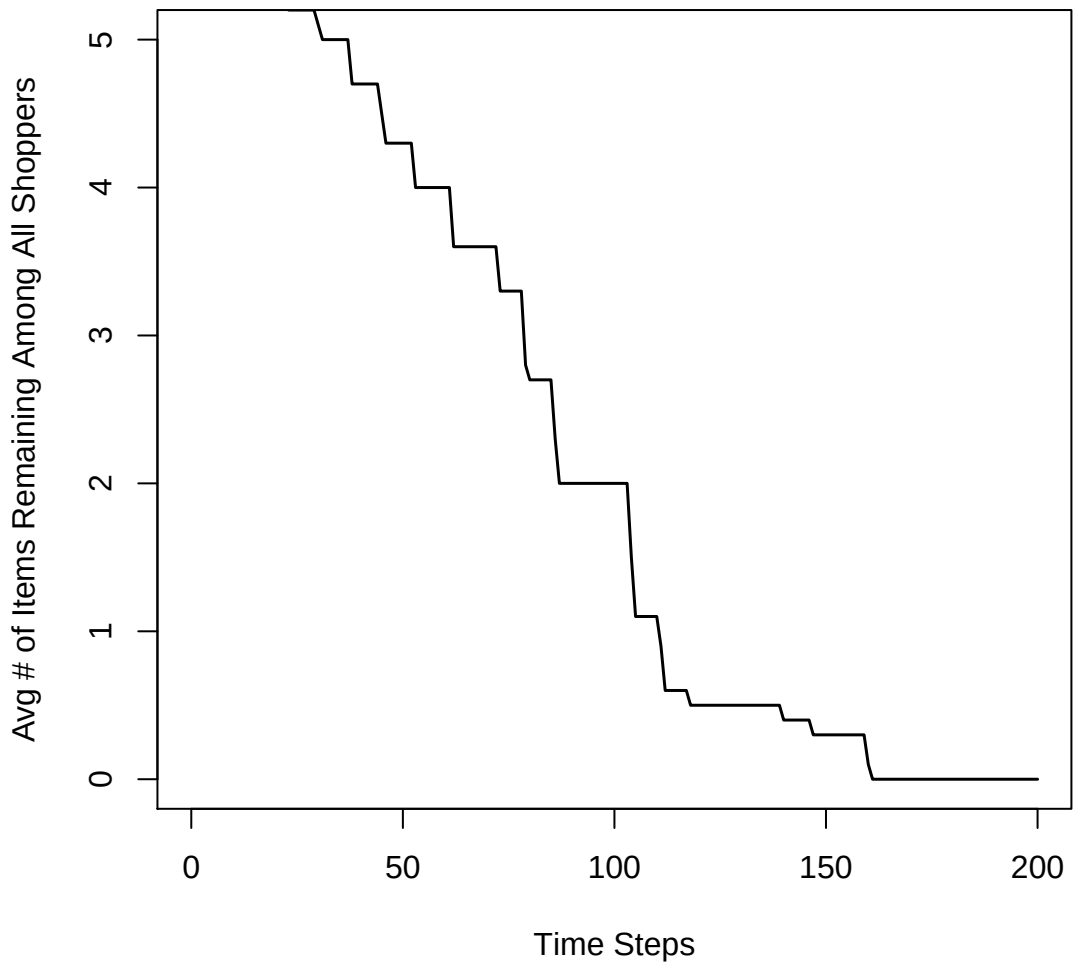
```
  time = time + 1
  } # <-- this one closes the time loop

master_past_storage[ , path ] = storage_num_items_remaining

}  # <-- this one closes the path loop
```

First let's look at what the result of a single path would look like:

```
plot( y = storage_num_items_remaining ,
      x = 0:(length(storage_num_items_remaining)-1) ,
      type ='l' , lwd = 1.5    ,
      xlab = "Time Steps" ,
      ylab = "Avg # of Items Remaining Among All Shoppers",
      ylim = c(0,5) )
```



Here we see that the average number of items decreases rapidly at first, which makes sense as shoppers find the "easy" items on their lists that are perhaps abundantly stocked by traders. However, around t=100, the series plateaus as shoppers struggle to find the remaining items on their lists.

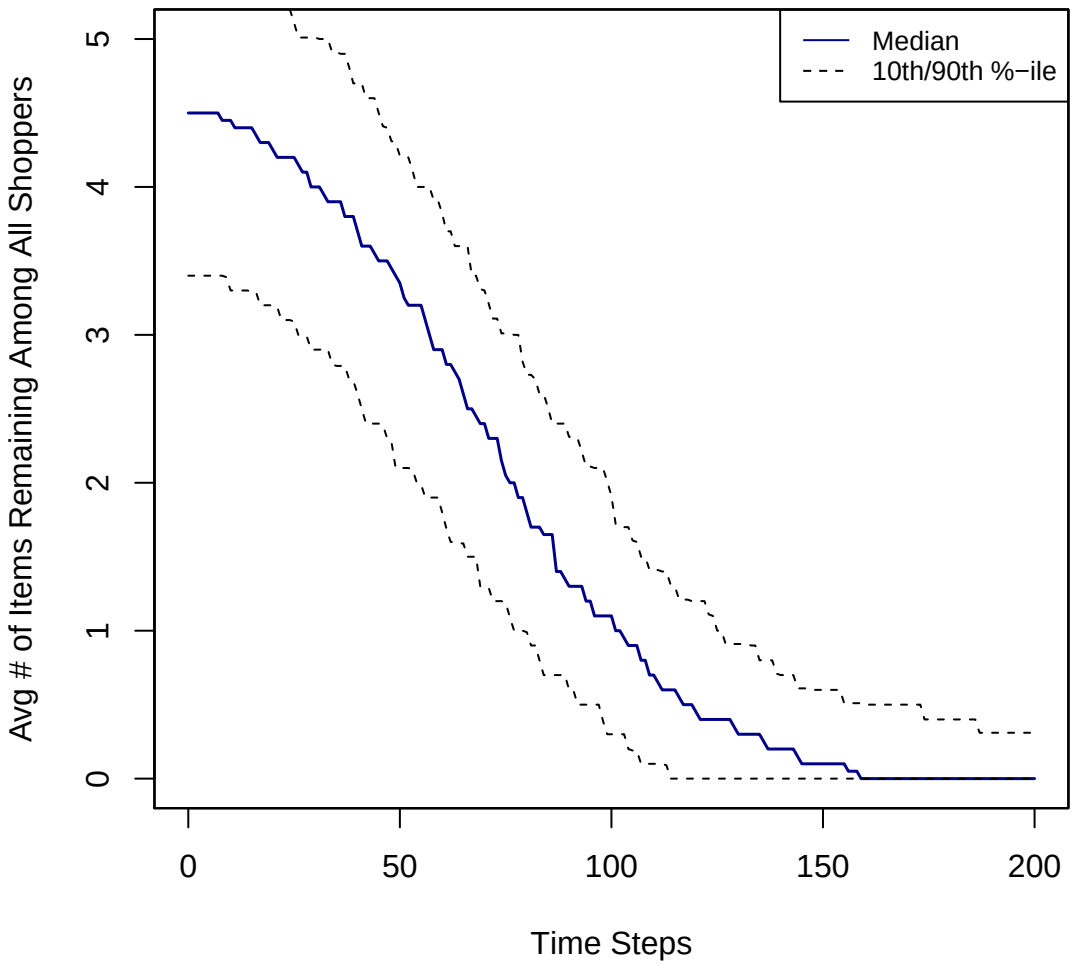
And, in fact, by the end of our set simulation time, the average number of items still required across shoppers has NOT dropped to zero. So, at the end of our simulation, we can say that there were still "unsatisfied" shoppers.

An important feature of the model to note here is that it *is* possible for a shopper to need an item that is not carried by *any* trader in the marketplace. And so, revisitng the prior comment, it is unclear if the shoppers with remaining list items simply have not found a given stall yet, or if the items they still require are not stocked by *any* trader in the marketplace (because of the random generation process for both shopper lists and trader stall inventories).

But the figure above was just for a single simulation path. Let's leverage the 100 paths that we simulated in order to get an idea of the summary statistics of multiple outcomes:

```
t = apply( master_past_storage , 1 , FUN = function(x){
  quantile( x , probs = c(0.1,.5,0.9) )
})

plot( y = t[2,] ,
      x = 0:(length(storage_num_items_remaining)-1) ,
      type ='l' , col = 'darkblue' ,
      lwd = 1.5   ,
      xlab = "Time Steps" ,
      ylab = "Avg # of Items Remaining Among All Shoppers",
      ylim = c(0,5) )
par(new=TRUE)
plot( y = t[1,] ,
      x = 0:(length(storage_num_items_remaining)-1) ,
      type ='l' , col = 'black' ,
      lwd = 1, lty = 2   ,
      xaxt = 'n' , yaxt='n' ,
      xlab = NA, ylab = NA,
      ylim = c(0,5) )
par(new=TRUE)
plot( y = t[3,] ,
      x = 0:(length(storage_num_items_remaining)-1) ,
      type ='l' , col = 'black' ,
      lwd = 1, lty = 2   ,
      xaxt = 'n' , yaxt='n' ,
      xlab = NA, ylab = NA,
      ylim = c(0,5) )
legend( 'topright', cex = 0.8,
        legend = c("Median","10th/90th %-ile"),
        col = c("darkblue","black"),
        lwd = c(1,1), lty = c(1,2) )
```



Here we could argue that there is a better average outcome: the median result (call this the "base case") is that all shoppers find everything they need, and typically well before the simulation is complete (say, around t=150).

In fact, the 10th Percentile ("optimistic case") outcome is that all shoppers are done closer to t=100.

Conclusion & Planned Extensions

As described throughout, there are multiple extensions that could be made to this model, but I'm pretty happ with this in terms of a decent starting point for demonstrating the capabilities and insights that can be derived from an Agent-Based Model.

For documentation, some extensions might be:

- Add pricing, so that each trader stall might charge a different price for the same product, and shoppers have a finite amount of money to spend.

- Incorporate inventories: in other words, traders can run out of goods that they sell. This would allow us to explore questions like optimal inventory levels for each trader.
- Track the position of each shopper at each point in time, so that we can plot "travel maps", which would help with the optimization of trader stall location.
- Make this into an interactive R Shiny app, where many of the constants defined at the start of the script are slider bars that can be adjusted by the user.