Reactive Stock Trading Strategies for Maximizing Returns and Mitigating Risk

Andrew Schneer
Math 87 – Mathematical Modeling
Professor James Adler
Final Project
12/12/2014

Abstract

A model was developed to simulate a purely reactive, fairly high-frequency stock trading strategy using Monte Carlo methods in order to determine how profitable the strategy would be. A virtual investment interface was modeled in Matlab, including a virtual portfolio, virtual exchange, virtual stocks, and a virtual investment account. These tools were used to feed both randomly generated stock data, and also real historical stock data from a major NYSE corporation, into the trading algorithm to gauge performance. The simulation was run over the course of 8,000 days with a trade frequency of approximately one trade per day. It was found that the trade commission paid to the broker, among other factors, significantly influenced the outcome of the simulation. With a $10 fixed per-trade fee, the total account and portfolio worth dropped steadily throughout. With no commission, the total worth rose steadily. There is likely a commission threshold over which money cannot be made and under which money can be made. This would be something to explore in future work.

The model was designed with the average investor in mind. This simulation is intended to be adaptable for a working person who has saved some money to be able to experiment with an investment strategy in order to plan their finances accordingly. The model would have to be altered significantly if it were to be used for large investment companies or algorithmic trading houses.

Introduction

The goal of this study was to model a purely reactive stock trading strategy and simulate it using both theoretical and true stock price data. Much of the mathematical modeling work that has been done with regards to stock markets around the world has been focused on trying to model either a stock price or the market as a whole. In one case, researchers were able to establish a model based on random buy and sell requests which could predict the average spread (difference between lowest ask price and highest bid price of a stock) for the London Stock Exchange with shockingly good accuracy[1]. A regression on the predicted average spread model and the true spread produced a coefficient of determination (R^2) of 0.96, meaning that the model is an excellent representation of the true spread behavior.

Additional research has been done on algorithmic trading (AT), but usually with the goal of understanding its holistic effect on the overall market. For example, researchers at UC-Berkley published two papers discussing the effects of AT on the overall market. The first investigates how AT affects the "price discovery process," and draws conclusions about how AT relates to liquidity and price volatility[2]. The second study analyzes AT in terms of whether or not it improves the liquidity and quality of the market. The researchers concluded that AT tends to narrow bid-ask spreads, improve liquidity, and cause stock quotes to be more representative of the actual values of the securities which they represent[3].

It is clear that information about specific algorithmic trading strategies is more difficult to find than overall market research, other than simple concepts and models. This makes sense because any algorithmic trading strategy that works is extremely valuable and worth keeping secret. Algorithmic trading houses spend thousands of dollars per year developing proprietary trading algorithms to stay ahead of the competition.

In the case of this study, modeling the market as a whole was not the intention. In addition, modeling stock price activity was only a small part of it. Some stock price data sets were created based on random variation and other techniques, but the focus was testing real data. A Matlab simulation was created which could be run using different sources of stock price data in order to determine whether or not the trading strategy under investigation would significantly increase investment returns over time. The trading strategy was implemented in the form of a Monte Carlo simulation, and it was run repeatedly on different sets of data. It was expected that the results of the simulation would support the trading strategy as a good method of improving returns.

Major constraints for this study included time, computational power, and accuracy. Developing a Matlab model that could effectively simulate the trading algorithm on large data sets and export the results in a form that was usable took a lot of creative program design and debugging. Even after the model was built, computational power became an issue because of the time it took to execute the simulations. Most importantly, however, was the accuracy constraint. Optimizing this model to be completely realistic and accurate was outside the scope of the project. Thus, the model was built in such a way that it was generic enough so parameters could be manipulated and it could be easily optimized and improved on in the future.

The Model

Overview:

Anyone who has every paid any attention to the stock market has probably heard someone say "buy low, sell high." But what does that actually mean? Depending on how it is interpreted, it can mean drastically different things. One interpretation is to try and predict when and how much a stock price will increase, and then try to buy it when it is at its lowest point. Another strategy might be to buy when the price starts going up with the hope that the purchase occurred on the early end of a large price increase. This study attempted to quantify "buy low, sell high" by creating a purely reactive trading algorithm which decides how much of a stock to buy and sell strictly based on the price change each day. In order to do this, a threshold is established for buying and selling shares of a particular stock. Every time the stock price changes and crosses the threshold, a calculated number of shares is either purchased or sold.

The construction of the model mimics the steps outlined by Scott Rothbort in his 2008 article in The Street[4]. The first step was to come up with a hypothesis about some factor in a stock price that could either be predicted or exploited to turn a profit. The major underlying assumption for this model was that the stock price would generally remain constant on the average. The hypothesis was that if the price was generally constant, a collection of small profits could be made by riding the small upswings and downswings in the price, which could add up to a significant amount of money in the long run. A fairly steady stock would be ideal for this type of strategy, however whether the stock price were to go up, down, or stay flat, this strategy would arguably make money along the way, either increasing returns or offsetting losses.

Parameters:

The model was set up in such a way that the user was given a set of parameters to directly control. This allowed the model to be tweaked and made it easier for further functionality to be added on later. The key parameters are summarized in Table 1. All of the data in this report was simulated using the default values in this table.

Parameter	Description	Default Value
TRADE_COMMISSION	Fixed commission to be paid to the broker per trade, regardless of the number of shares being traded.	**$10.00**
MIN_TRANS_PROFIT	Minimum amount of revenue that a buy transaction would have to produce in addition to covering the trade commission.	**$5.00**
STOCK_AVG_WINDOW	Number of previous days over which to average the stock price. This was used for the non-linear threshold calculation method, since it required calculation of the threshold based on current data, not data on day zero.	**100 Days (or all days if less than 100)**
ACCOUNT_BALANCE_INIT	Amount of money that starts off in the investment account at the beginning of the simulation.	**$10,000**
INITIAL_PURCHASE_AMOUNT	Amount of money from the investment account allocated for initially purchasing the stock at the beginning of the simulation.	**$5,000 (half of initial account balance)**
Stock price simulation method	Method used to provide stock price data for running the simulation.	• **Random** • **Real stock data**
Buy/sell threshold calculation method	Method used to determine the buy/sell thresholds for the simulation. These thresholds were given in number of shares to buy/sell per dollar of price change.	• **Non-linear**

Table 1: Key simulation parameters, descriptions, and values used.

The stock price simulation methods used were very different. The real stock data method simply pulled data for a specific stock from a pre-created CSV file containing stock price history data queried from Yahoo! Finance. The random stock data function, on the other hand, determined the stock price based on a Gaussian pseudo-random number distribution. Minimum

and maximum percent changes were declared, and then the Gaussian distribution was transformed to match the 0-σ and 5-σ points with the two percent change limits. This way, the minimum percent change was the most likely (0-σ), the largest percentage change was the least likely (5-σ), and any percent change less than the minimum, including zero change, was impossible.

The buy/sell threshold calculation functions were broken into either linear or non linear strategies. The linear strategy based the threshold on the original price and original number of shares. The threshold was given by equation (1).

$$Threshold_{Linear} = \frac{Original \ \ \#Shares \ on \ Day \ Zero}{Original \ \ Price \ on \ Day \ Zero} \quad (1)$$

The non-linear threshold was calculated according to equation (2).

$$Threshold_{Non-linear} = \frac{Number \ \ of \ shares \ currently \ \ owned}{Price \ \ average \ \ over \ \ last \ \text{"avgWindow"} days} \quad (2)$$

The reason why the linear method was linear was because the ratio was always constant. For the non-linear method on the other hand, the threshold would increase as more shares were purchased and the average price dropped, while it would decrease as more shares were sold and the average price decreased. Thus, the amount of shares being sold in response to a price change would become increasingly larger as the price deviated from the average.

Data Structures:

The model was set up with a series of custom structure data types in Matlab. Structures were defined for a stock, a portfolio, an investment account, and a performance database. These data structures were used to keep track of different properties and data so as to mimic their roles in a real investment situation.

The stock exchange was set up as an array of stock structures. Each stock structure contained information pertaining to a stock in the exchange, such as its name, symbol, current price, and lists of the high, low, and close data for the stock each day since the simulation was started. The exchange array was populated with all the stocks that would be tested in a particular run of the simulation.

Similarly, a portfolio structure was created to keep track of the virtual trader's stock portfolio. This structure contained data such as the last date a trade was made in the entire portfolio, a list of the stock symbols and corresponding number of shares for each stock owned, the total current investment, revenue, and value for the portfolio, and a complete list of all transactions occurring since the beginning of the simulation. The transactions were kept in a table with data such as the type of transaction (buy/sell), stock symbol, number of shares, price, timestamp of the transaction, and total value of the transaction.

An investment account structure was used to keep track of the money that was not tied up in the portfolio. In order to accurately determine the long term return on a high-frequency-traded investment, it was necessary to have a means of adding up and storing the gains/losses from all the small transactions, while accounting for a fixed trade commission.

Finally, the performance database structure was used to keep track of the portfolio over time. The portfolio data structure was set up in such a way that it was volatile. In other words, its properties were changed every time a transaction occurred, and the old data was not saved.

The performance structure essentially logged the data in the portfolio structure for each day of trading to keep track of it throughout the course of the simulation.

Algorithms:

The basic workflow of the program is outlined in Figure 1. The procedure was to initialize a stock in the exchange, buy an initial number of shares, and then loop through the simulation phase many times. During the simulation phase, on each day of trading, the stock's new price was generated based on either a custom function or on real historical stock data, and then the portfolio was updated by determining how many shares of the stock should be bought or sold based on the new price. Once the portfolio was updated, all the new data was loaded into the appropriate data structures, the transactions were recorded to the transaction history, and the cycle repeated for as many days as were specified by the user. Upon completion of the simulation, the data was exported to CSV files for analysis in Excel.

The most significant and interesting algorithms used in the simulation were those used for calculating the buy/sell thresholds, deciding how to react to a price change, and generating pseudo-random stock price activity.

Two strategies were attempted for determining the buy/sell thresholds; one linear and one non-linear. The linear version based the thresholds on a stock's price and number of shares owned on day zero, when the simulation started.

Assumptions:

Perhaps the most important aspect of this study to consider is the assumptions that were made when building the model. In this model, a significant assumption was that the stock price would stay relatively constant over time. This assumption was used to create functions to randomly simulate stock price over time. The random stock price functions used Gaussian distributed pseudo-random numbers to decide what the new stock price should be as a percentage increase or decrease from the stock price on day zero. This strategy allowed the stock price to vary randomly while forcing larger price changes to be much less likely to occur than smaller ones. This assumption did not apply when real stock data was used, however it still represents the ideal type of stock that would be expected to do well with this investment strategy and keep the risk low. The constant-average assumption is important because regardless of how much money can be made from doing quick buys and sells, if the average stock price plummets, then the investor loses all of his/her money anyway. Thus, a random stock price simulator that always tends to return to some average value is not realistic. There are other factors that influence the price beyond pure chance.

Another important assumption that was made was that every transaction request was processed instantaneously. This is unrealistic because in any type of free market, there has to be a buyer in order for there to be a seller, and vice versa. In reality, a buy/sell request is submitted; this is also called a bid/ask. The stock exchange attempts to set up buyers with sellers based on their bid/ask prices. When a real buy or sell is actually executed, often the price at which it occurs varies slightly from the initial bid or ask.

Making the "perfect transaction" assumption could impact the model in the following ways. First, if actual transaction prices are slightly different than how they appear in the model, then it could impact the buy/sell decision making. Since the model uses precise threshold calculations to determine how many shares to buy or sell, any small change in the price could

either keep it under or force it over the threshold when it would not otherwise. This could change the entire course of decision making by the program, leading to a completely different outcome regarding investment returns.

Probably the most accurate assumption was that the trade commission would be a fixed value. Almost all (if not all) stock brokerage houses, especially online, use fixed trade commissions. This means that they charge a fixed price per transaction, regardless of the number of shares being bought or sold. The only time this assumption becomes inaccurate is if very large volumes are traded with very large sums of money. In this case, brokerage houses will sometimes offer discounts for high-volume traders. However, this is not a critical issue because most ordinary people probably do not trade nearly frequently enough to warrant a commission discount. If this factor were to be taken into account, it would involve simply creating a function that would calculate the trade commission based on the number of shares being transacted.

Another assumption that was made involves the symmetry of the buy and sell price thresholds. The buy and sell thresholds were assumed to be the same, that is symmetrical. In other words, the amount of price increase necessary to warrant a sell request was the same as the amount of price decrease necessary to warrant a buy request. This turned out to be the simplest and most logical way to implement the model. The reason why no other method was tested is that it did not make sense to do so. Any aspect of the model that could potentially disrupt the symmetry of the stock price, and either cause the price or the buy/sell behavior to act in a biased manner, would have been foolish to implement in the first phase of testing. In addition, there was no basis for thinking that it either represented true market behavior, or that it would add anything valuable to the model. This factor can easily be tested in future simulations by simply modifying the threshold calculation function to set the buy and sell thresholds to two different values.

The stock price was always assumed to change. The random stock price generating function was written in such a way that any percent change in price lower than the minimum limit, including zero change, was impossible. This is unrealistic because theoretically a change of zero would be possible, but zero price change of a stock over the course of an entire day of trading has probably never happened, and thus it did not make sense to account for it in the model.

Sensitivity:

There were several key parameters in the model that contributed to its sensitivity; some more than others. These parameters included the trade commission, the minimum transaction profit, the stock price averaging window, and the initial number of shares purchased. The trade commission could potentially contribute a lot of sensitivity to the model because it is directly involved in the decision of whether or not to buy/sell, and how much. The minimum transaction profit could contribute sensitivity for the same reason. These two values work very closely to determine the buy/sell threshold.

The stock price averaging window would be unlikely to contribute sensitivity to the model. A change from 100 days of averaging to 110 would not change the average a lot. The larger the pool of numbers being averaged, the smaller a slight change in that number would affect the average.

The initial number of shares purchased could have some affect on the model, but probably not a lot. The initial number of shares does contribute to determining the buy/sell

thresholds, and also determines how much stock is available to play with. However, starting with slightly more or slightly fewer shares would not greatly influence the behavior of the simulation. It could however impact the total amount of profit made in the end. Since different investors have different amounts of money to invest, this is not really a factor that is vital to explore. There is no need to prove that more initial investment could lead to more overall return, because that is already a well-known fact. Tweaking this value would be more useful for simply conducting an accurate simulation for a particular investor whose finances are known so planning how to invest those finances would be as accurate as possible.

Results

For the purpose of this report, a simulation was run using 8,000 days of historical stock data for General Mills, Inc. (NYSE:GIS). Figure 2 highlights the performance of the portfolio and investment account relative to the stock price.

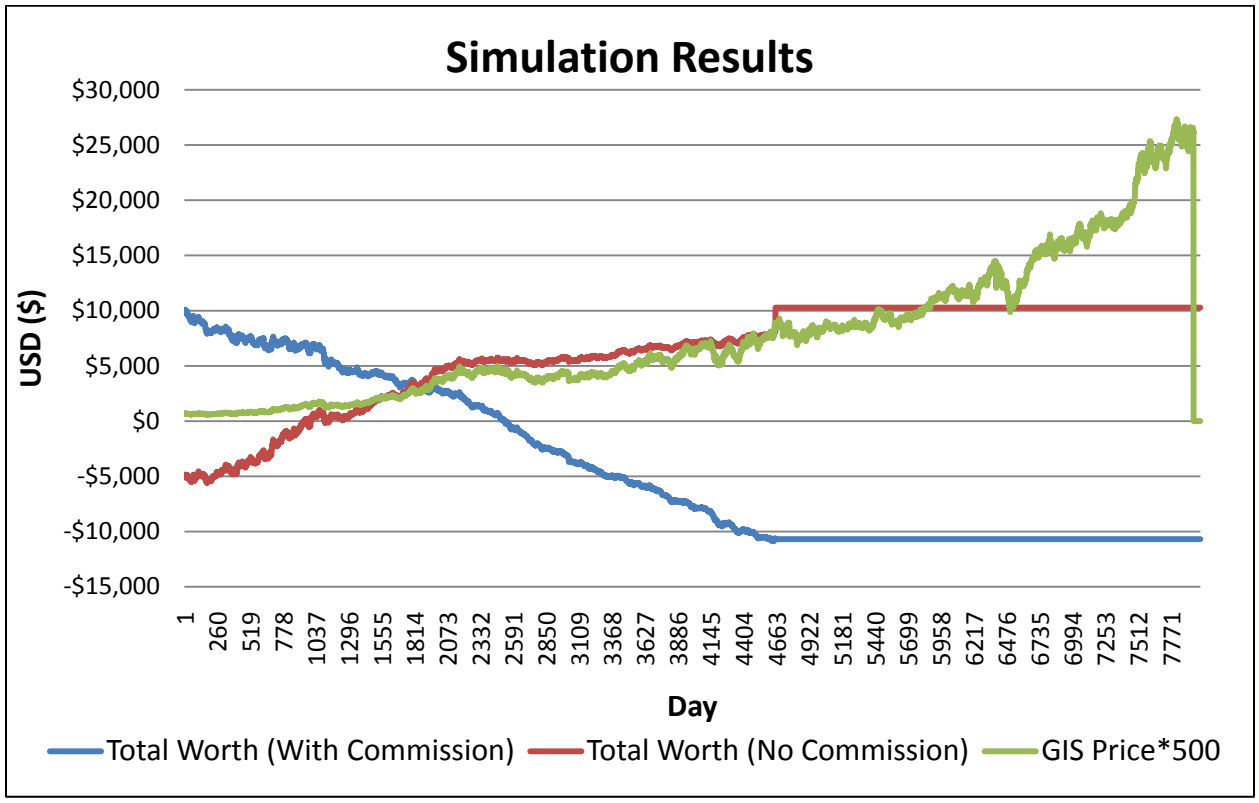

Figure 2: Simulation results for General Mills, Inc. (NYSE:GIS) with and without trade commission taken into account.

The data in Figure 2 shows a very interesting fact. The stock price commission seems to have a huge affect on whether or not money can be made with this model. When the commission was present, the total worth of the account and portfolio dropped steadily throughout the simulation. Without the commission, on the other hand, it thrived. This makes sense

because in order to make high frequency trades (once per day in this case), commission fees have to be sacrificed. There is obviously a tradeoff between commission fees and trade frequency, which is something that could definitely be explored in future simulations.

Something else that is notable is how the plot flattens out at about day 4663 and never bounces back into activity. The reason for this is likely that the stock price increased too much and all of the shares were sold off. One drawback of the model is that it is fairly static. In other words, a more sophisticated model would have better techniques for tracking the average price and growth patterns of the stock over time in order to continually recalculate parameters to optimize profit. In this model, the shares will never be bought back again unless the price drops down closer to where it was when the last shares were sold off.

Conclusions

A model was developed for testing a purely reactive stock trading strategy. Different parameters were tweaked to create as realistic a model as possible, and the simulation was run using both random stock data, and real stock data from Yahoo! Finance. Due to time constraints, only one simulation was ultimately set up and run for the purpose of thorough analysis. The total worth of the portfolio and investment account was logged over the course of 8,000 days of trading, and it was compared to the price of the stock as it varied over those same 8,000 days. It was found that the trade commission was much more influential in determining the profitability of the purely reactive strategy than originally suspected. Based on the simulation that was run, money would be steadily lost of a commission was present. If no commission was present, the portfolio would thrive. This opened up a lot of factors to test in future simulations, including running a wider array of commission values to see the results, and also testing other parameters as described in Table 1 to gauge their effects on the success of the investment.

Future Work

Beyond the results obtained from this study, there are many more interesting behaviors that can be explored using the same model. One of these is the transaction response time, which was discussed earlier. Implementing a correction factor to account for slight deviations in the actual buy/sell price relative to the bid/ask price would be simple, and would make the model more realistic.

Inflation is another thing that could be accounted for. It would be easy to add 3% annual depreciation to the model in order to produce a more realistic investment return forecast. Asymmetrical buy/sell thresholds could also shed some light on whether or not the trading strategy could be improved. Other methods for calculating the buy/sell thresholds, and also generating random stock price data would be worth exploring.

Beyond these things, the most important correlations that could be tested would be between the parameters listed in Table 1. Varying any combination of these factors while keeping the others constant would definitely produce valuable data for studying how the reactive trading strategy behaves. In addition, a time threshold could be calculated over the course of the simulation in which the risk of staying in the investment could be determined. For example, assuming the strategy makes money over time, the longer the investor stays with it, the more money they will make. As a result, the longer the investor stays in it successfully, the farther the

average stock price would have to drop before the investor gets back to where he/she started. For this reason, this type of investment strategy arguably becomes less risky over time. A graphical representation of this long-term risk mitigation would be a very interesting and useful task to pursue in the future.

References:

[1] J. D. FARMER, P. PATELLI, I. I. ZOVKO, *The predictive power of zero intelligence in financial markets*, PNAS 102 (6) 2254-2259 (2005), pp. 2254-2259

[2] T. HENDERSHOTT, R. RIORDAN, *Algorithmic Trading and Information*, 2009.

[3] T. HENDERSHOTT, C. M. JONES, A. J. MENKVELD, *Does Algorithmic Trading Improve Liquidity?*, J. Finance VOL. LXVI, NO. 1 (2011), pp. 1-34

[4] S. ROTHBORT, *How to Build Your Own Trading Model in 8 Steps*, The Street (2008), http://www.thestreet.com/story/10414044/1/how-to-build-your-own-trading-model-in-8-steps.html

Appendix

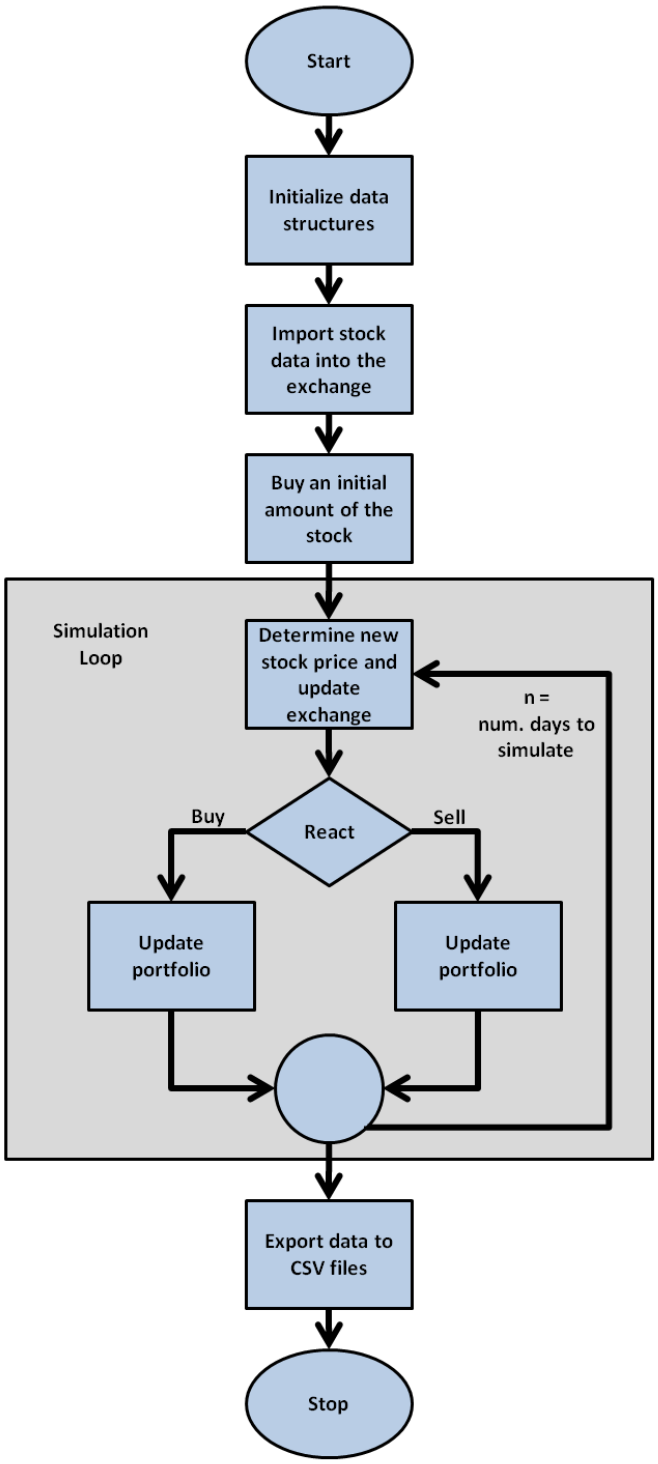

Figure 1: Monte Carlo simulation flowchart for reactive stock trading model.

```
Loading parameters...
Creating portfolio...
Creating performance database...
Creating investment account...
Current time is 12/12/2014 14:50:3.843100e+01
Creating stock exchange...
Buying initial stocks...

Simulation started...
Now simulating day: 1   of  8...
Now simulating day: 2   of  8...
Now simulating day: 3   of  8...
Now simulating day: 4   of  8...
Now simulating day: 5   of  8...
Now simulating day: 6   of  8...
Now simulating day: 7   of  8...
Now simulating day: 8   of  8...

Simulation Complete!
Exporting data to folder "./Simulation_..."
Exiting...

>>
```

```
function [flag,msg,portfolioMod,accountMod] = buy( ...
    portfolio,exchange,account, ...
    symbol,numShares,commission,timeStamp)

    % This function will "buy" the desired
    % number of shares of the desired stock,
    % referred to by its symbol.  This means
    % the stock will be added to the list of
    % stocks stored in the portfolio specified.

    % flag = 1 denotes success.
    % flag = 0 denotes failure.

    % Make sure more than zero
    % shares are being bought.
    % Otherwise exit.
    if(numShares <= 0)
        flag = 0;
        msg = 'Invalid number of shares!\n';
        portfolioMod = portfolio;
        accountMod = account;
        return;
    end

    % Check to see if the desired
    % stock is already present in the
    % portfolio.  "strcmp()" will return
    % a matrix of 1s or 0s corresponding
    % to the "symbol" being compared
    % to each element in the cell array
    % "portfolio.stockSymbols".  That
    % matrix is then searched with "find()"
    % for a "1", which will indicate the
    % index of that match in the stockSymbols
    % matrix.
    temp = strcmp(portfolio.stockSymbols,symbol);
    I = find((temp==1),1,'first');
    % Case where the stock is already in the
    % portfolio.
    if(~isempty(I))
        % Add shares to the
        % shares already owned.
        portfolio.stockShares(I) = ...
            portfolio.stockShares(I) + numShares;
    else
        % Add the stock to the list
        % of stocks in the portfolio.
        nextStockNum = (length(portfolio.stockSymbols) + 1);
        portfolio.stockSymbols(nextStockNum) = {symbol};
        portfolio.stockShares(nextStockNum) = numShares;
    end
```

```
% Record the date of the buy as the
% last day of trading to date.
portfolio.lastTradeDay_year = timeStamp(1);
portfolio.lastTradeDay_month = timeStamp(2);
portfolio.lastTradeDay_day = timeStamp(3);

% Record information about the transaction.

% Find the next empty row in the
% transaction history list.
nextTransNum = (size(portfolio.transactions,1) + 1);
% Add the transaction information
% to the list.
portfolio.transactions{nextTransNum,1} = 'BUY';
portfolio.transactions{nextTransNum,2} = timeStamp(1);
portfolio.transactions{nextTransNum,3} = timeStamp(2);
portfolio.transactions{nextTransNum,4} = timeStamp(3);
portfolio.transactions{nextTransNum,5} = timeStamp(4);
portfolio.transactions{nextTransNum,6} = timeStamp(5);
portfolio.transactions{nextTransNum,7} = timeStamp(6);
portfolio.transactions{nextTransNum,8} = symbol;
[flag,tempStock] = getStockData_exchange(exchange,symbol);
if(flag == 0)
    % Error.  Stock not found in exchange.
    flag = 0;
    msg = 'Stock not in exchange.';
    portfolioMod = portfolio;
    accountMod = account;
    return;
else
    portfolio.transactions{nextTransNum,9} ...
        = tempStock.currentPrice;
end
portfolio.transactions{nextTransNum,10} = numShares;
portfolio.transactions{nextTransNum,11} = ...
    (tempStock.currentPrice * numShares);

% Update the portfolio with newly
% calculated values.  a, b, and c are
% dummies, just care about getting the
% updated portfolio back from the function.
[a,b,c,portfolio] = calcInvestment(portfolio,exchange);

% Update the investment account
% by adding/subtracting from
% the balance.
nextIndex = (length(account.year) + 1);
account.year(nextIndex) = timeStamp(1);
account.month(nextIndex) = timeStamp(2);
account.day(nextIndex) = timeStamp(3);
```

```
    account.balance(nextIndex) =...
        (account.balance(nextIndex-1) ...
        - (tempStock.currentPrice * numShares) ...
        - commission);

    % Make sure to return the newly
    % modified portfolio struct
    % and account struct.
    portfolioMod = portfolio;
    accountMod = account;

    flag = 1;
    msg = 'Success!';
    return;

end
```

```
function [totalInvestment,...
            totalRevenue,...
            totalValue,...
            portfolioMod]...
            = calcInvestment(...
                portfolio,...
                exchange)

    % This function will search
    % through the transaction
    % history of the specified
    % portfolio and add up all the
    % buy and sell values to determine
    % the total amount of money that
    % has been invested and the total
    % that has been returned from sales.
    % This will also calculate the total
    % value of the portfolio at the present
    % time.

    totalInvestment = 0;
    totalRevenue = 0;
    totalValue = 0;

    % Calculate total investment and
    % total revenue.
    for i = (1:size(portfolio.transactions,1))
        if(strcmp(portfolio.transactions(i,1),'BUY'))
            totalInvestment = totalInvestment...
                + portfolio.transactions{i,11};
        elseif(strcmp(portfolio.transactions(i,1),'SELL'))
            totalRevenue = totalRevenue...
                + portfolio.transactions{i,11};
        else
            % Error.
        end
    end

    % Calculate the total value of the
    % portfolio.

    % Loop through all the stock symbols in
    % the portfolio.
    for i = (1:length(portfolio.stockSymbols))
        % Search exchange for corresponding symbol.
        [flag,currentStock] = getStockData_exchange(...
            exchange,portfolio.stockSymbols(i));
        if(flag == 0)
            % Error.  Stock not found in exchange.
            totalInvestment = -1;
            totalRevenue = -1;
```

```
            totalValue = -1;
            return;
        else
            totalValue = totalValue ...
                + (currentStock.currentPrice ...
                * portfolio.stockShares(i));
        end
    end

    % Save information to the portfolio.
    portfolio.totalInvestment = totalInvestment;
    portfolio.totalRevenue = totalRevenue;
    portfolio.totalValue = totalValue;

    % Make sure to return the
    % updated portfolio struct.
    portfolioMod = portfolio;

    return;

end
```

```
function [flag,thresh] = calcTransThresh_01(...
                exchange,...
                portfolio,...
                symbol)
    % This function calculates
    % the buy/sell threshold
    % for a stock using a linear
    % strategy.  In other words,
    % the amount by which the
    % price has deviated from the
    % average will NOT affect
    % the buy/sell threshold.
    % No matter how high or low
    % the price goes, the same
    % number of shares will be
    % bought/sold per $ of change.

    % This function assumes that
    % the long term average of the
    % stock in question is equal to
    % the price of the stock when
    % the first shares were first
    % purchased (on day zero).

    % The threshold will be given
    % in number of shares per dollar
    % to either buy or sell given
    % the change in stock price.
    % If the function succeeds,
    % flag = 1, otherwise it
    % is set to 0.

    % Get stock data from the exchange.
    [flag,stockStruct] = getStockData_exchange(...
        exchange,symbol);
    % Make sure the stock is found.
    if(flag == 0)
        % Handle error.
        fprintf('Stock NOT found in exchange!\n');
        thresh = 0;
        flag = 0;
        return;
    end
    % Get the average stock price
    % for the given stock over time,
    % which in the case of this
    % function will be given by
    % the price when the stock was
    % initially purchased on day zero.
    avgPrice = stockStruct.close(1);
    % Get the initial number of
```

```
% shares from the initial purchase
% on day zero.
for i = (1:size(portfolio.transactions,1))
    if(((portfolio.transactions{i,4} == 0) ...
            || (portfolio.transactions{i,4} == 1)) ...
            && strcmp(portfolio.transactions{i,8},symbol))
        startShares = portfolio.transactions{i,10};
        break;
    end
    % If control reaches this point,
    % the first-day transaction for
    % the desired symbol was not found.
    % Handle error.
    fprintf('Start shares not found in portfolio!\n');
    flag = 0;
    thresh = 0;
    return;
end
% Determine the buy/sell threshold
% by dividing the original number
% of shares purchased by the initial
% price when the shares were first
% purchased.  This threshold is chosen
% based on the assumption that the
% buy and sell thresholds will be
% equal.  Thus, the threshold
% calculated here represents the
% amount of price change that would
% have to occur to either wipe out
% or double the number of shares
% owned relative to the original number
% of shares owned, assuming
% the price changes for those two
% scenarios would be equal.
thresh = (startShares / avgPrice);

% Note that this threshold represents
% a continuous slope of shares vs. price.
% In other words, going strictly by this
% number, an infinitesimal price change
% will warrant the purchase/sale of
% a corresponding infinitesimal number
% of shares.  This is unrealistic
% because, firstly, transactions take
% time to execute, and if a
% transaction were to be required
% for every infinitesimal price change,
% then the transactions would be
% required to execute
% instantaneously.  Also, because
% of trade commissions, this buy/sell
```

```
    % strategy would be not cost effective
    % by a long shot.  In order to deal
    % with this issue, the continuous-slope
    % threshold will be used as a baseline
    % threshold, and then an additional
    % calculation will take into account
    % a fixed trade commission.  A
    % transaction will not be made unless
    % enough profit would be made on the
    % sale to cover the commission.  If
    % the transaction is a buy, the same
    % threshold will be used as if it were
    % a sell, since money is always lost
    % on a buy transaction.  Essentially,
    % the model will traverse the
    % shares/price slope calculated here
    % until it makes sense to actually
    % execute a transaction.

    % Return success flag.
    flag = 1;

    return;
end
```

```
function [flag,thresh] = calcTransThresh_02(...
                exchange,...
                portfolio,...
                symbol,...
                avgWindow)
    % This function calculates
    % the buy/sell threshold
    % for a stock using a non-linear
    % strategy.  In other words,
    % the amount by which the
    % price has deviated from the
    % average WILL affect
    % the buy/sell threshold.
    % The buy/sell threshold
    % will be larger the farther
    % the price deviates from the
    % average during the last
    % "x" days.  This means the
    % simulation will buy/sell
    % more shares for each $
    % of price change as the
    % price deviates more and more
    % from the average.

    % The threshold will be given
    % in number of shares per dollar
    % to either buy or sell given
    % the change in stock price.
    % If the function succeeds,
    % flag = 1, otherwise it
    % is set to 0.

    % Get stock data from the exchange.
    [flag,stockStruct] = getStockData_exchange(...
        exchange,symbol);
    % Make sure the stock is found.
    if(flag == 0)
        % Handle error.
        fprintf('Stock NOT found in exchange!\n');
        thresh = 0;
        flag = 0;
        return;
    end
    % Get the average stock price
    % for the given stock over time,
    % which in the case of this
    % function will be given by
    % the average of the price
    % over the last "x" days.
    if(length(stockStruct.close) <= avgWindow)
        % Average all elements of
```

```
    % "close" array.
    avgPrice = mean(stockStruct.close);
else
    % Average last "x" elements
    % of "close" array.
    temp = (length(stockStruct.close) ...
            - avgWindow + 1);
    avgPrice = mean(...
        stockStruct.close(temp:end));
end;
% Get the number of shares
% currently owned for the
% desired symbol.
temp = strcmp(portfolio.stockSymbols,symbol);
I = find((temp == 1),...
        1,'first');
currentShares = portfolio.stockShares(I);
% Determine the buy/sell threshold
% by dividing the current number
% of shares purchased by the average
% price over the last "x" days.
% This threshold is chosen
% based on the assumption that the
% buy and sell thresholds will be
% equal.  Thus, the threshold
% calculated here represents the
% amount of price change that would
% have to occur to either wipe out
% or double the number of shares
% owned relative to the current number
% of shares owned, assuming
% the price changes for those two
% scenarios would be equal.
thresh = (currentShares / avgPrice);

% Note that this threshold represents
% a continuous slope (non-linear) of
% shares vs. price.
% In other words, going strictly by this
% number, an infinitesimal price change
% will warrant the purchase/sale of
% a corresponding infinitesimal number
% of shares.  This is unrealistic
% because, firstly, transactions take
% time to execute, and if a
% transaction were to be required
% for every infinitesimal price change,
% then the transactions would be
% required to execute
% instantaneously.  Also, because
% of trade commissions, this buy/sell
```

```
    % strategy would be not cost effective
    % by a long shot.  In order to deal
    % with this issue, the continuous-slope
    % threshold will be used as a baseline
    % threshold, and then an additional
    % calculation will take into account
    % a fixed trade commission.  A
    % transaction will not be made unless
    % enough profit would be made on the
    % sale to cover the commission.  If
    % the transaction is a buy, the same
    % threshold will be used as if it were
    % a sell, since money is always lost
    % on a buy transaction.  Essentially,
    % the model will traverse the
    % shares/price slope calculated here
    % until it makes sense to actually
    % execute a transaction.

    % Return success flag.
    flag = 1;

    return;
end
```

```
function [emptyAccount] = createEmptyAccount( ...
                                name)

    % Definition of structure "performance".

    emptyAccount = struct( ...
        'name',                 '',...
        'year',                 [],...
        'month',                [],...
        'day',                  [],...
        'balance',              []);

    emptyAccount.name = name;

    return;

end
```

```
function [emptyPerformance] = createEmptyPerformance( ...
                                name,portfolioTracked)

    % Definition of structure "performance".

    emptyPerformance = struct( ...
        'name',                 '',...
        'portfolioTracked',     '',...
        'year',                 [],...
        'month',                [],...
        'day',                  [],...
        'totalInvestment',      [],...
        'totalRevenue',         [],...
        'totalValue',           []);

    emptyPerformance.name = name;
    emptyPerformance.portfolioTracked = ...
        portfolioTracked;

    return;

end
```

```
function [emptyPortfolio] = createEmptyPortfolio(name)

    % Definition of structure "portfolio".

    % The time here is of the last day
    % of trading of any stock
    % in the portfolio.  Each one is
    % a single value, not a matrix.

    % Matrix of the value of the portfolio
    % at the time of closing each day
    % recorded in the list.

    % This will be a matrix containing
    % a table of transaction information
    % (stock, numShares, date, time,
    % buy/sell, price, etc).  Each row
    % will be a transaction.  This matrix
    % will be easily exported to a CSV
    % file later.

    % The "stocks" matrix has the symbols
    % of all the stocks in the portfolio.

    % The "transactions" matrix has a table
    % of all the data for each transaction.
    % The columns, from left to right, are:
        % 01    transaction ('BUY' or 'SELL')
        % 02    year
        % 03    month
        % 04    day
        % 05    hour
        % 06    minute
        % 07    second
        % 08    stock symbol
        % 09    price
        % 10    number of shares
        % 11    total

    emptyPortfolio = struct(...
        'name',                 '',...
        'lastTradeDay_year',    0,...
        'lastTradeDay_month',   0,...
        'lastTradeDay_day',     0,...
        'stockSymbols',         {{}},...
        'stockShares',          [],...
        'totalInvestment',      0,...
        'totalRevenue',         0,...
        'totalValue',           0,...
        'transactions',         {{}});
```

```
    emptyPortfolio.name = name;

    return;

end
```

```
function [emptyPortfolioStockData] = createEmptyPortfolioStockData()

    % Definition of structure "portfolioStockData".

    % The time here is of the last day
    % of trading of the particular
    % stock.

    % Matrix of the value of the portfolio
    % at the time of closing each day
    % recorded in the list.

    % There will be a matrix containing
    % a table of transaction information
    % (stock, numShares, date, time,
    % buy/sell, price, etc).  Each row
    % will be a transaction.  This matrix
    % will be easily exported to a CSV
    % file later.

    % The "transactions" matrix has a table
    % of all the data for each transaction.
    % The columns, from left to right, are:
        % 01    transaction ('BUY' or 'SELL')
        % 02    year
        % 03    month
        % 04    day
        % 05    hour
        % 06    minute
        % 07    second
        % 08    stock symbol
        % 09    price
        % 10    number of shares
        % 11    total

    emptyPortfolioStockData = struct(...
        'symbol',               '',...
        'numShares',            0,...
        'transactions',         {{}});

    return;

end
```

```
function [emptyStock] = createEmptyStock()

    % Definition of structure "stock".

    % Each field is a matrix of values
    % corresponding to the different
    % data for the day described by the
    % "year", "month", and "day" matrices.

    emptyStock = struct(...
        'name',          [],...
        'symbol',        [],...
        'exchange',      [],...
        'currentPrice', 0,...
        'year',          [],...
        'month',         [],...
        'day',           [],...
        'high',          [],...
        'low',           [],...
        'close',         [],...
        'volume',        []);

    return;

end
```

```
function [exchangeMatrix] = createExchange_01(...
                                timeStamp)

    % Create theoretical stocks.

    % This will create a list of
    % theoretical stocks with information
    % about their behavior.

    % Create theoretical stocks and add data
    % to their structures.  Create a
    % "stockExchange" matrix to store a list
    % of all the stocks theoretically available
    % for purchase.  Each stock in the exchange
    % will have

    % This code will create 5 imaginary stocks
    % with a range of share prices.  The
    % stock exchange, timestamp, and trading
    % volume will be the same for all the stocks,
    % since price is the focus of the simulation.

    exchangeMatrix(1) = createStock(...
        'Stock_A',...
        'AAA',...
        'Exchange_01',...
        10.0,...
        timeStamp(1),...
        timeStamp(2),...
        timeStamp(3),...
        11.0,...
        9.0,...
        10.0,...
        10000);
    exchangeMatrix(2) = createStock(...
        'Stock_B',...
        'BBB',...
        'Exchange_01',...
        50.0,...
        timeStamp(1),...
        timeStamp(2),...
        timeStamp(3),...
        51.0,...
        49.0,...
        50.0,...
        10000);
    exchangeMatrix(3) = createStock(...
        'Stock_C',...
        'CCC',...
        'Exchange_01',...
        90.0,...
```

```
        timeStamp(1),...
        timeStamp(2),...
        timeStamp(3),...
        91.0,...
        89.0,...
        90.0,...
        10000);
    exchangeMatrix(4) = createStock(...
        'Stock_D',...
        'DDD',...
        'Exchange_01',...
        130.0,...
        timeStamp(1),...
        timeStamp(2),...
        timeStamp(3),...
        131.0,...
        129.0,...
        130.0,...
        10000);
    exchangeMatrix(5) = createStock(...
        'Stock_E',...
        'EEE',...
        'Exchange_01',...
        170.0,...
        timeStamp(1),...
        timeStamp(2),...
        timeStamp(3),...
        171.0,...
        169.0,...
        170.0,...
        10000);

    return;

end
```

```
function [exchangeMatrix] = createExchange_02( ...
                                timeStamp)

    % Create theoretical stocks.

    % This will create a list of
    % theoretical stocks with information
    % about their behavior.

    % Create theoretical stocks and add data
    % to their structures.  Create a
    % "stockExchange" matrix to store a list
    % of all the stocks theoretically available
    % for purchase.  Each stock in the exchange
    % will have

    % This code will create 5 imaginary stocks
    % with a range of share prices.  The
    % stock exchange, timestamp, and trading
    % volume will be the same for all the stocks,
    % since price is the focus of the simulation.

    exchangeMatrix(1) = createStock( ...
        'Stock_A',...
        'AAA',...
        'Exchange_01',...
        50.0,...
        timeStamp(1),...
        timeStamp(2),...
        timeStamp(3),...
        51.0,...
        49.0,...
        50.0,...
        10000);

    return;

end
```

```
function [exchangeMatrix] = createExchange_03(...
                                timeStamp)

    % Create theoretical stocks.

    % This will create a list of
    % theoretical stocks with information
    % about their behavior.

    % Create theoretical stocks and add data
    % to their structures.  Create a
    % "stockExchange" matrix to store a list
    % of all the stocks theoretically available
    % for purchase.  Each stock in the exchange
    % will have

    % For this version of the function,
    % pull initial data directly from
    % Yahoo! Finance historical data.

    stockSymbol = 'GIS';
    filename = sprintf('./Historical_Stock_Data/%s.csv',...
                       stockSymbol);
    dataAll = csvread(filename);
    dataClose = dataAll(:,4);
    startPrice = dataClose(1);

    % Create exchange matrix.
    exchangeMatrix(1) = createStock(...
        'General_Mills',...
        'GIS',...
        'NYSE',...
        startPrice,...
        timeStamp(1),...
        timeStamp(2),...
        timeStamp(3),...
        -1,...
        -1,...
        startPrice,...
        -1);

    return;

end
```

```
function [exchangeMatrix] = createExchange_04( ...
                            stockToSimulate, ...
                            symbolToSimulate, ...
                            exchangeToSimulate, ...
                            timeStamp)

    % Create theoretical stocks.

    % This will create a list of
    % theoretical stocks with information
    % about their behavior.

    % Create theoretical stocks and add data
    % to their structures.  Create a
    % "stockExchange" matrix to store a list
    % of all the stocks theoretically available
    % for purchase.  Each stock in the exchange
    % will have

    % For this version of the function,
    % pull initial data directly from
    % Yahoo! Finance historical data.

    stockSymbol = symbolToSimulate;
    filename = sprintf('./Historical_Stock_Data/%s.csv',...
                       stockSymbol);
    dataAll = csvread(filename);
    dataClose = dataAll(:,4);
    startPrice = dataClose(1);

    % Create exchange matrix.
    exchangeMatrix(1) = createStock( ...
        stockToSimulate, ...
        symbolToSimulate, ...
        exchangeToSimulate, ...
        startPrice, ...
        timeStamp(1), ...
        timeStamp(2), ...
        timeStamp(3), ...
        -1, ...
        -1, ...
        startPrice, ...
        -1);

    return;

end
```

```
function [stockStruct] = createStock(...
                name,...
                symbol,...
                exchange,...
                currentPrice,...
                year,...
                month,...
                day,...
                high,...
                low,...
                close,...
                volume)

    % Function to "create a stock".

    % This is a theoretical stock.
    % The "creation" means that the
    % stock structure is filled with
    % initial data which defines it.

    % This returns a stock structure
    % with all the initial information
    % about that stock's current value
    % and last day of trading in it.

    % The date information refers to the
    % date when the last trading occurred
    % for the stock.  The price and trade
    % information is for that last day
    % of trading.

    % Create stock structure.
    stockStruct = createEmptyStock();

    % Modify stock structure and
    % add all the relevant information.
    stockStruct.name = name;
    stockStruct.symbol = symbol;
    stockStruct.exchange = exchange;
    stockStruct.currentPrice = currentPrice;
    stockStruct.year(1) = year;
    stockStruct.month(1) = month;
    stockStruct.day(1) = day;
    stockStruct.high(1) = high;
    stockStruct.low(1) = low;
    stockStruct.close(1) = close;
    stockStruct.volume(1) = volume;

    return;

end
```

```
function [] = exportData(...
            exchange,...
            portfolio,...
            performance,...
            account,...
            timeStamp)
    % Export all data from the
    % simulation to files that can
    % be analyzed in Excel.
    % The data will be written
    % to text files, but separated
    % by commas to satisfy the CSV
    % format.

    % Simulation output data - folder name:
    timeStampString = sprintf('%04d%02d%02d_%02d_%02d_%02.0f',...
        timeStamp(1),timeStamp(2),timeStamp(3), ...
        timeStamp(4),timeStamp(5),timeStamp(6));
    % Concatenate strings to make folder name.
    outputFolderName = ['./Simulation_' timeStampString];

    % Create folder for output data.
    mkdir(outputFolderName);

    % Write portfolio performance data
    % to a CSV file.
    % Prepare file.
    temp = sprintf('/%s.csv',performance.name);
    fileName_performance = [outputFolderName temp];
    fileID_performance = fopen(fileName_performance, 'w');
    % Write data.
    fprintf(fileID_performance,'Performance Struct Name:,');
    fprintf(fileID_performance,performance.name);
    fprintf(fileID_performance,'\n');
    fprintf(fileID_performance,'Portfolio Tracked:,');
    fprintf(fileID_performance,performance.portfolioTracked);
    fprintf(fileID_performance,'\n\n');
    fprintf(fileID_performance,'Year,');
    fprintf(fileID_performance,'Month,');
    fprintf(fileID_performance,'Day,');
    fprintf(fileID_performance,'Total Investment,');
    fprintf(fileID_performance,'Total Revenue,');
    fprintf(fileID_performance,'Total Value,');
    fprintf(fileID_performance,'\n');
    for i = (1:length(performance.year))
        fprintf(fileID_performance,'%d,',performance.year(i));
        fprintf(fileID_performance,'%d,',performance.month(i));
        fprintf(fileID_performance,'%d,',performance.day(i));
        fprintf(fileID_performance,'%0.2f,',performance.totalInvestment(i));
        fprintf(fileID_performance,'%0.2f,',performance.totalRevenue(i));
        fprintf(fileID_performance,'%0.2f',performance.totalValue(i));
```

```
        fprintf(fileID_performance,'\n');
    end
    % Close file.
    fclose(fileID_performance);

    % Write investment account data
    % to a CSV file.
    % Prepare file.
    temp = sprintf('/%s.csv',account.name);
    fileName_account = [outputFolderName temp];
    fileID_account = fopen(fileName_account,'w');
    % Write data.
    fprintf(fileID_account,'Account Name:,');
    fprintf(fileID_account,account.name);
    fprintf(fileID_account,'\n\n');
    fprintf(fileID_account,'Year,');
    fprintf(fileID_account,'Month,');
    fprintf(fileID_account,'Day,');
    fprintf(fileID_account,'Balance,');
    fprintf(fileID_account,'\n');
    for i = (1:length(account.year))
        fprintf(fileID_account,'%d,',account.year(i));
        fprintf(fileID_account,'%d,',account.month(i));
        fprintf(fileID_account,'%d,',account.day(i));
        fprintf(fileID_account,'%0.2f',account.balance(i));
        fprintf(fileID_account,'\n');
    end
    % Close file.
    fclose(fileID_account);

    % Write transaction table to a file.
    % Prepare file.
    fileName_transactions = [outputFolderName  '/transactions.csv'];
    fileID_transactions = fopen(fileName_transactions,'w');
    % Write data.
    fprintf(fileID_transactions,'Transaction,');
    fprintf(fileID_transactions,'Year,');
    fprintf(fileID_transactions,'Month,');
    fprintf(fileID_transactions,'Day,');
    fprintf(fileID_transactions,'Hour,');
    fprintf(fileID_transactions,'Minute,');
    fprintf(fileID_transactions,'Second,');
    fprintf(fileID_transactions,'Symbol,');
    fprintf(fileID_transactions,'Price,');
    fprintf(fileID_transactions,'Shares,');
    fprintf(fileID_transactions,'Total');
    fprintf(fileID_transactions,'\n');
    for i = (1:size(portfolio.transactions,1))
        fprintf(fileID_transactions,'%s,',portfolio.transactions{i,1});
        fprintf(fileID_transactions,'%d,',portfolio.transactions{i,2});
        fprintf(fileID_transactions,'%d,',portfolio.transactions{i,3});
```

```
        fprintf(fileID_transactions,'%d,',portfolio.transactions{i,4});
        fprintf(fileID_transactions,'%d,',portfolio.transactions{i,5});
        fprintf(fileID_transactions,'%d,',portfolio.transactions{i,6});
        fprintf(fileID_transactions,'%0.3f,',portfolio.transactions{i,7});
        fprintf(fileID_transactions,'%s,',portfolio.transactions{i,8});
        fprintf(fileID_transactions,'%0.2f,',portfolio.transactions{i,9});
        fprintf(fileID_transactions,'%d,',portfolio.transactions{i,10});
        fprintf(fileID_transactions,'%0.2f',portfolio.transactions{i,11});
        fprintf(fileID_transactions,'\n');
    end
    % Close file.
    fclose(fileID_transactions);

    % Write stock exchange data to a file.
    % Prepare folder.
    stockExchangeDataFolder = [outputFolderName  '/Stock_Exchange'];
    mkdir(stockExchangeDataFolder);
    % Create a data file for each stock
    % in the exchange.
    for i = (1:length(exchange))
        % Prepare current data file.
        currentStockDataFile = sprintf('/%s.csv',exchange(i).symbol);
        fileName_currentStock =...
            [stockExchangeDataFolder currentStockDataFile];
        fileID_currentStock = fopen(fileName_currentStock, 'w');
        % Write data.
        fprintf(fileID_currentStock,'Name,');
        fprintf(fileID_currentStock,'%s\n',exchange(i).name);
        fprintf(fileID_currentStock,'Symbol,');
        fprintf(fileID_currentStock,'%s\n',exchange(i).symbol);
        fprintf(fileID_currentStock,'Exchange,');
        fprintf(fileID_currentStock,'%s\n',exchange(i).exchange);
        fprintf(fileID_currentStock,'Current Price,');
        fprintf(fileID_currentStock,'%0.2f\n',exchange(i).currentPrice);
        fprintf(fileID_currentStock,'\n');
        fprintf(fileID_currentStock,'Stock Activity');
        fprintf(fileID_currentStock,'\n\n');
        fprintf(fileID_currentStock,'Year,');
        fprintf(fileID_currentStock,'Month,');
        fprintf(fileID_currentStock,'Day,');
        fprintf(fileID_currentStock,'High,');
        fprintf(fileID_currentStock,'Low,');
        fprintf(fileID_currentStock,'Close,');
        fprintf(fileID_currentStock,'Volume');
        fprintf(fileID_currentStock,'\n');
        for j = (1:length(exchange(i).year))
            fprintf(fileID_currentStock,'%d,',exchange(i).year(j));
            fprintf(fileID_currentStock,'%d,',exchange(i).month(j));
            fprintf(fileID_currentStock,'%d,',exchange(i).day(j));
            fprintf(fileID_currentStock,'%0.2f,',exchange(i).high(j));
            fprintf(fileID_currentStock,'%0.2f,',exchange(i).low(j));
```

```
            fprintf(fileID_currentStock,'%0.2f,',exchange(i).close(j));
            fprintf(fileID_currentStock,'%d',exchange(i).volume(j));
            fprintf(fileID_currentStock,'\n');
        end
        % Close current file.
        fclose(fileID_currentStock);
    end

    % Copy the parameters file
    % into the results folder
    % so we know which parameters
    % were used to produce that
    % data.
    copyfile('./parameters.m',outputFolderName);

    return;

end
```

```
function [flag,stockStruct] = getStockData_exchange( ...
                              exchange,symbol)
    % This function will search the
    % given exchange matrix for the
    % desired stock.  If it is found,
    % the stock struct is returned
    % and the flag is "1".
    % If not, then "0" is returned
    % as the flag and the "stockStruct"
    % is returned empty.

    for i = (1:length(exchange))
        if(strcmp(exchange(i).symbol,symbol))
            stockStruct = exchange(i);
            flag = 1;
            return;
        end
    end
    % If desired stock is not found,
    % return 0.
    stockStruct = createEmptyStock();
    flag = 0;
    return;
end
```

```
function [flag,portfolioStockData] = getStockData_portfolio( ...
                            portfolio,symbol)
    % This function will search the
    % given portfolio for the
    % desired stock.  If it is found,
    % the stock info is returned
    % and the flag is "1".
    % If not, then "0" is returned
    % as the flag and the stockInfo
    % variables are returned as
    % either zero or empty.
    % The data that is returned
    % for the stock (if it is found
    % in the portfolio) is:
    %   1.  Name of stock
    %   2.  Symbol of stock
    %   3.  Last trade year, month,
    %       and day for the stock
    %   4.  Number of shares of that
    %       stock currently in the
    %       portfolio
    %   5.  List of all transactions
    %       for that stock to date
    % The data will be in the form of
    % a "portfolioStockData" struct.

    % Create an empty portfolioStockData
    % struct.
    portfolioStockData = createEmptyPortfolioStockData();
    % Search the portfolio for the
    % desired stock.
    for i = (1:length(portfolio.stockSymbols))
        if(strcmp(portfolio.stockSymbols(i),symbol))
            % If stock is found, save
            % all the necessary data to the
            % struct to be returned.
            portfolioStockData.symbol = portfolio.stockSymbols(i);
            portfolioStockData.numShares = portfolio.stockShares(i);
            % Search through the transactions list
            % for all transactions involving the
            % desired stock.  Copy them into the
            % list to be returned.
            k = 1;
            for j = (1:size(portfolio.transactions,1))
                if(strcmp(portfolio.transactions{j,8},symbol))
                    portfolioStockData.transactions(k,:) = ...
                        portfolio.transactions(j,:);
                    k = k + 1;
                end
            end
            % Return success.  All data is
```

```
            % ready to be returned.
            flag = 1;
            return;
        end
    end
    % If desired stock is not found,
    % return 0.
    portfolioStockData = createEmptyPorftolioStockData();
    flag = 0;
    return;
end
```

```
function [endTime] = incrementTime(...
                    startTime,increment)

    % Function to increment a
    % date vector by a specified
    % amount of years, months, days,
    % hours, minutes, and seconds.

    % Date vector looks like this:
    % [Y M D H MN S].

    endTime = zeros(1,6);

    % Increment seconds.
    new_sec = startTime(6) + increment(6);
    increment(6) = increment(6) ...
        + mod(new_sec,60);
    increment(5) = increment(5) ...
        + floor(new_sec/60);

    % Increment minutes.
    new_min = startTime(5) + increment(5);
    increment(5) = increment(5) ...
        + mod(new_min,60);
    increment(4) = increment(4) ...
        + floor(new_min/60);

    % Increment hours.
    new_hour = startTime(4) + increment(4);
    increment(4) = increment(4) ...
        + mod(new_hour,24);
    increment(3) = increment(3) ...
        + floor(new_hour/24);

    % month_28.25 = (startTime(2) == 2);
    % month_30 = ((startTime(2) == 4)...
    %   || (startTime(2) == 6)...
    %   || (startTime(2) == 9)...
    %   || (startTime(2) == 11));
    % month_31 = ((startTime(2) == 1)
    %   || (startTime(2) == 3)...
    %   || (startTime(2) == 5)...
    %   || (startTime(2) == 7)...
    %   || (startTime(2) == 8)...
    %   || (startTime(2) == 10)...
    %   || (startTime(2) == 12));

    % % Increment days.
    % new_day = startTime(3) + increment(3);
    % if(new_day > )
    % increment(3) = increment(3)...
```

```
    %   + mod(new_day,???????????);
    % increment(2) = increment(2)...
    %   + floor(new_day/?????????????);

    % Increment months.

    % Increment years.
%   endTime(1) = startTime(1) + increment(1);

    return;

end
```

```
%% MAIN FILE 01:

% Units are $=USD.

% This is the main start-file for the
% random reactive stock simulation.

% Clear Freemat environment.
clear all;
close all;
clc;

% Print a few blank lines.
fprintf(1,'\n');

% Include external files.
% Because of how these files
% are written, variables are
% created with these names.
% Don't reuse these names in
% future code.
fprintf('Loading parameters...\n');
parameters;

% Create a portfolio to keep track of
% the stocks invested in.
fprintf('Creating portfolio...\n');
Portfolio_01 = createEmptyPortfolio('Portfolio_01');

% Create a performance struct to keep
% track of Portfolio_01 data over time.
fprintf('Creating performance database...\n');
Performance_01 = createEmptyPerformance(...
    'Performance_01','Portfolio_01');

% Set starting time and date for
% the simulation.
timeStamp = clock;
fprintf('Current time is %02d/%02d/%04d %02d:%02d:%02d\n',...
    timeStamp(2),timeStamp(3),timeStamp(1),...
    timeStamp(4),timeStamp(5),timeStamp(6));

% Set the starting day of the
% simulation to "1".  Rather than
% keeping track of the calendar
% dates, for now the transaction
% dates will simply be "days after
% day zero".
startDate = [0 0 1 0 0 0];

% Day zero (for initializing
```

```
% stock exchange).
dayZero = zeros(1,6);

% Create a stock exchange from where
% stocks can be "purchased" and added
% to the portfolio.  This stock
% exchange will only change when
% the function "updateExchange"
% modifies it.
fprintf('Creating stock exchange...\n');
Exchange_01 = createExchange(...
                    dayZero);

fprintf('Buying initial stocks...\n');
% Add initial stocks (hypothetical).
% Create several imaginary stocks that
% cover a wide range of prices, from
% very cheap to very expensive (per share).
[flag,msg,Portfolio_01] = buy(Portfolio_01,Exchange_01, ...
    'AAA',100,dayZero);
if(flag == 0)
    % Error.  Buy failed.
    fprintf(1,'\nError:  Buy failed.\n');
    fprintf(1,msg);
    fprintf(1,'\n');
    return;
end
[flag,msg,Portfolio_01] = buy(Portfolio_01,Exchange_01, ...
    'BBB',100,dayZero);
if(flag == 0)
    % Error.  Buy failed.
    fprintf(1,'\nError:  Buy failed.\n');
    fprintf(1,msg);
    fprintf(1,'\n');
    return;
end
[flag,msg,Portfolio_01] = buy(Portfolio_01,Exchange_01, ...
    'CCC',100,dayZero);
if(flag == 0)
    % Error.  Buy failed.
    fprintf(1,'\nError:  Buy failed.\n');
    fprintf(1,msg);
    fprintf(1,'\n');
    return;
end
[flag,msg,Portfolio_01] = buy(Portfolio_01,Exchange_01, ...
    'DDD',100,dayZero);
if(flag == 0)
    % Error.  Buy failed.
    fprintf(1,'\nError:  Buy failed.\n');
    fprintf(1,msg);
```

```
    fprintf(1,'\n');
    return;
end
[flag,msg,Portfolio_01] = buy(Portfolio_01,Exchange_01, ...
    'EEE',100,dayZero);
if(flag == 0)
    % Error.  Buy failed.
    fprintf(1,'\nError:  Buy failed.\n');
    fprintf(1,msg);
    fprintf(1,'\n');
    return;
end

% RUN SIMULATION:

fprintf('\nSimulation started...\n');

% This loop runs for a specific
% amount of time.  It modifies the
% properties of the stocks in the
% exchange.  Then the portfolio
% automatically decides how to react
% depending on the conditions set forth
% in the "parameters" file.  Each time
% a transaction is made, it is written
% to the portfolio struct and eventually
% exported to a CSV file.

% Duration in the form of
% Years, Months, Days, Hours,
% Minutes, Seconds.  Only
% use durations in DAYS
% until a function is written
% for tracking the actual
% calendar date.
duration = [0 0 SIMULATION_DAYS 0 0 0];
%               incrementTime(duration);
endDate = startDate + duration;
% Adjust endDate so last day is
% equal to the value of the duration,
% assuming day 1 is the first day.
endDate(3) = endDate(3) - 1;

% Uncomment this line to activate
% code to clear previous print
% line in real time.
%       reverseStr = '';

% Record initial performance
% of Portfolio_01.
Performance_01 = updatePerformance(...
```

```
    Performance_01,...
    Portfolio_01,...
    zeros(1,6));

% Simulation loop.
for i = ((startDate(3)-1):(endDate(3)-1))

    % The simulation will have a
    % resolution of 1 day.  That is,
    % the simulated market will only
    % change/update once per day, at
    % which time the program will decide
    % how to react.  The first day of the
    % simulation will be day "0".

    % Keep track of the current day.
    currentDay = zeros(1,6);
    currentDay(3) = (startDate(3) + i);

    % Print feedback to the user to track
    % simulation progress.
    msg = sprintf('Now simulating day:\t%d\tof\t%d...\n',...
        currentDay(3),endDate(3));
    fprintf(1,msg);
% Attempt to clear previous print line
% in real time.  Found code online.
% Uncomment to turn it back on, but
% it doesn't work.
%   fprintf(1,[reverseStr msg]);
%   reverseStr = repmat(sprintf('\b'), 1, length(msg));

    % Update the exchange with simulated
    % market activity.


    % FIX THIS SO THAT STOCK PRICE
    % IS MORE STATIC AND SO THAT IT
    % NEVER GOES BELOW ZERO.


    Exchange_01 = updateExchange(Exchange_01,currentDay);
    % Update the portfolio by executing
    % buy and sell transactions based
    % on the market activity for the
    % last day in question.
    Portfolio_01 = updatePortfolio(Exchange_01, ...
        Portfolio_01,currentDay, ...
        TRADE_COMMISSION, ...
        MIN_TRANS_PROFIT, ...
        STOCK_AVG_WINDOW);
    % Save current portfolio data to
```

```
    % the performance struct.
    Performance_01 = updatePerformance( ...
        Performance_01, ...
        Portfolio_01, ...
        currentDay);
end

fprintf(1,'\nSimulation Complete!\n');

% Export data to files.
fprintf(1,'Exporting data to folder "./Simulation_..."\n' );
exportData(Exchange_01, ...
    Portfolio_01, ...
    Performance_01, ...
    timeStamp);

fprintf(1,'Exiting...\n\n');

return;
```

```
%% MAIN FILE 02:

% Units are $=USD.

% This is the main start-file for the
% random reactive stock simulation.

% Clear Freemat environment.
clear all;
close all;
clc;

% Print a few blank lines.
fprintf(1,'\n');

% Include external files.
% Because of how these files
% are written, variables are
% created with these names.
% Don't reuse these names in
% future code.
fprintf('Loading parameters...\n');
parameters;

% Create a portfolio to keep track of
% the stocks invested in.
fprintf('Creating portfolio...\n');
Portfolio_01 = createEmptyPortfolio('Portfolio_01');

% Create a performance struct to keep
% track of Portfolio_01 data over time.
fprintf('Creating performance database...\n');
Performance_01 = createEmptyPerformance(...
    'Performance_01','Portfolio_01');

% Create an investment account to keep
% track of investment amount and returns.
fprintf('Creating investment account...\n');
Account_01 = createEmptyAccount(...
    'Account_01');

% Set starting time and date for
% the simulation.
timeStamp = clock;
fprintf('Current time is %02d/%02d/%04d %02d:%02d:%02d\n',...
    timeStamp(2),timeStamp(3),timeStamp(1), ...
    timeStamp(4),timeStamp(5),timeStamp(6));

% Set the starting day of the
% simulation to "1".  Rather than
% keeping track of the calendar
```

```
% dates, for now the transaction
% dates will simply be "days after
% day zero".
startDate = [0 0 1 0 0 0];

% Day zero (for initializing
% stock exchange).
dayZero = zeros(1,6);

% Initialize investment
% account info.
Account_01.year(1) = dayZero(1);
Account_01.month(1) = dayZero(2);
Account_01.day(1) = dayZero(3);
Account_01.balance(1) = ACCOUNT_BALANCE_INIT;

% Create a stock exchange from where
% stocks can be "purchased" and added
% to the portfolio.  This stock
% exchange will only change when
% the function "updateExchange"
% modifies it.
fprintf('Creating stock exchange...\n');
Exchange_01 = createExchange_04(...
                NAME_TO_SIMULATE,...
                SYMBOL_TO_SIMULATE,...
                EXCHANGE_TO_SIMULATE,...
                dayZero);

fprintf('Buying initial stocks...\n');
% Add initial stocks to the portfolio.
symbolToPurchase = SYMBOL_TO_SIMULATE;
[flag,stock] = getStockData_exchange(...
    Exchange_01,...
    symbolToPurchase);
if(flag == 0)
    % Handle error.
    fprintf('Failed to retrieve stock data from exchange.\n');
    return;
end
sharesToPurchase = floor(...
    INITIAL_PURCHASE_AMOUNT / stock.currentPrice);
% Buy.
[flag,msg,Portfolio_01,Account_01] = buy(...
    Portfolio_01,...
    Exchange_01,...
    Account_01,...
    symbolToPurchase,...
    sharesToPurchase,...
    TRADE_COMMISSION,...
    dayZero);
```

```
if(flag == 0)
    % Error.  Buy failed.
    fprintf(1,'\nError:  Buy failed.\n');
    fprintf(1,msg);
    fprintf(1,'\n');
    return;
end

% RUN SIMULATION:

fprintf('\nSimulation started...\n');

% This loop runs for a specific
% amount of time.  It modifies the
% properties of the stocks in the
% exchange.  Then the portfolio
% automatically decides how to react
% depending on the conditions set forth
% in the "parameters" file.  Each time
% a transaction is made, it is written
% to the portfolio struct and eventually
% exported to a CSV file.

% Duration in the form of
% Years, Months, Days, Hours,
% Minutes, Seconds.  Only
% use durations in DAYS
% until a function is written
% for tracking the actual
% calendar date.
duration = [0 0 SIMULATION_DAYS 0 0 0];
%               incrementTime(duration);
endDate = startDate + duration;
% Adjust endDate so last day is
% equal to the value of the duration,
% assuming day 1 is the first day.
endDate(3) = endDate(3) - 1;

% Uncomment this line to activate
% code to clear previous print
% line in real time.
%       reverseStr = '';

% Record initial performance
% of Portfolio_01.
Performance_01 = updatePerformance(...
    Performance_01,...
    Portfolio_01,...
    dayZero);

% Simulation loop.
```

```
for i = ((startDate(3)-1):(endDate(3)-1))

    % The simulation will have a
    % resolution of 1 day.  That is,
    % the simulated market will only
    % change/update once per day, at
    % which time the program will decide
    % how to react.  The first day of the
    % simulation will be day "0".

    % Keep track of the current day.
    currentDay = zeros(1,6);
    currentDay(3) = (startDate(3) + i);

    % Print feedback to the user to track
    % simulation progress.
    msg = sprintf('Now simulating day:\t%d\tof\t%d...\n',...
        currentDay(3),endDate(3));
    fprintf(1,msg);
% Attempt to clear previous print line
% in real time.  Found code online.
% Uncomment to turn it back on, but
% it doesn't work.
%   fprintf(1,[reverseStr msg]);
%   reverseStr = repmat(sprintf('\b'), 1, length(msg));

    % Update the exchange with simulated
    % market activity.


    % FIX THIS SO THAT STOCK PRICE
    % IS MORE STATIC AND SO THAT IT
    % NEVER GOES BELOW ZERO.


    Exchange_01 = updateExchange(...
        Exchange_01,...
        currentDay);
    % Update the portfolio by executing
    % buy and sell transactions based
    % on the market activity for the
    % last day in question.
    [Portfolio_01,Account_01] = updatePortfolio(...
        Exchange_01,...
        Portfolio_01,...
        Account_01,...
        currentDay,...
        TRADE_COMMISSION,...
        MIN_TRANS_PROFIT,...
        STOCK_AVG_WINDOW);
    % If account hasn't been updated,
```

```
    % update it.  This is to ensure
    % that even if there isn't a buy/sell,
    % the account still gets updated so
    % it has continuous data for
    % graphing later against other data.
    if(Account_01.day(end) ~= currentDay(3))
        % Update the investment account
        % by adding/subtracting from
        % the balance.
        nextIndex = (length(Account_01.year) + 1);
        Account_01.year(nextIndex) = currentDay(1);
        Account_01.month(nextIndex) = currentDay(2);
        Account_01.day(nextIndex) = currentDay(3);
        Account_01.balance(nextIndex) = ...
            Account_01.balance(nextIndex-1);
    else
        % Do nothing.  The account
        % has already been updated
        % for the current day.
    end
    % Save current portfolio data to
    % the performance struct.
    Performance_01 = updatePerformance( ...
        Performance_01, ...
        Portfolio_01, ...
        currentDay);
end

fprintf(1,'\nSimulation Complete!\n');

% Export data to files.
fprintf(1,'Exporting data to folder "./Simulation_..."\n' );
exportData(Exchange_01, ...
    Portfolio_01, ...
    Performance_01, ...
    Account_01, ...
    timeStamp);

fprintf(1,'Exiting...\n\n');

return;
```

```
%% Parameters:

% SET PARAMETERS FOR THE
% SIMULATION HERE.  THEN,
% RUN "main.m".  MAKE SURE
% ALL CODE FILES ARE IN THE
% SAME FOLDER AND THAT
% FOLDER IS OPEN IN FREEMAT/
% MATLAB.


% NOTE:  TO CHANGE THE
% STRATEGY USED FOR CALCULATING
% THE STOCK PRICE BEHAVIORS,
% SEE THE "updateExchange()"
% FUNCTION.


            % % This is the amount of
            % % revenue that would be
            % % earned from selling
            % % the given stock at the
            % % new price relative to
            % % its price the day before.
            % THRESHOLD_SELL = 15;
            % % This represents the amount
            % % of money that would be
            % % saved by buying the stock
            % % at the current price
            % % relative to the price
            % % the day before.
            % THRESHOLD_BUY = 15;



SIMULATION_DAYS = 8000;
% This is the commission
% charged by the broker
% per trade in USD/trade.
TRADE_COMMISSION = 0;
% This is the minimum
% required profit (gain
% minus commission) required
% to justify a sell, in $.
MIN_TRANS_PROFIT = 5;
% Averaging window for
% function that determines
% the average of the stock
% price, in # days.
STOCK_AVG_WINDOW = 100;
% Account starting balance
```

```
% in $.
ACCOUNT_BALANCE_INIT = 10000;
% How much of account should
% be spent on initial stock
% purchase.
INITIAL_PURCHASE_AMOUNT =...
    (ACCOUNT_BALANCE_INIT / 2);
% Stock name to purchase
% and simulate.
NAME_TO_SIMULATE = 'General_Mills';
% Stock symbol to purchase
% and simulate.
SYMBOL_TO_SIMULATE = 'GIS';
% Exchange of stock being
% simulated.
EXCHANGE_TO_SIMULATE = 'NYSE';
```

```
function [flag,msg,portfolioMod,accountMod] = sell( ...
    portfolio,exchange,account, ...
    symbol,numShares,commission,timeStamp)

    % This function will "sell" the desired
    % number of shares of the desired stock,
    % referred to by its symbol.  This means
    % the stock might be removed to the list of
    % stocks stored in the portfolio specified.

    % flag = 1 denotes success.
    % flag = 0 denotes failure.

    % Make sure more than zero
    % shares are being sold.
    % Otherwise exit.
    if(numShares <= 0)
        flag = 0;
        msg = 'Invalid number of shares!\n';
        portfolioMod = portfolio;
        accountMod = account;
        return;
    end

    % Check to see if the desired
    % stock is already present in the
    % portfolio.  "strcmp()" will return
    % a matrix of 1s or 0s corresponding
    % to the "symbol" being compared
    % to each element in the cell array
    % "portfolio.stockSymbols".  That
    % matrix is then searched with "find()"
    % for a "1", which will indicate the
    % index of that match in the stockSymbols
    % matrix.
    temp = strcmp(portfolio.stockSymbols,symbol);
    I = find((temp==1),1,'first');
    % Case where the stock is already in the
    % portfolio.
    if(~isempty(I))
        % Check to see if there are enough
        % shares in the portfolio to satisfy
        % the sell order.
        if(portfolio.stockShares(I) >= numShares)
            portfolio.stockShares(I) ...
                = portfolio.stockShares(I) - numShares;
            if(portfolio.stockShares(I) == 0)
                % If the zero shares of the
                % stock are currently owned,
                % DON'T remove it from the
                % portfolio.  This way its
```

```
                % data will be saved and
                % it can be bought again
                % under the same conditions
                % for buying if the price goes
                % down.

% % Remove the stock from the portfolio.
% portfolio.stockSymbols(I) = {};
% portfolio.stockShares(I) = [];

            end
        else
            % Error.  Can't sell more shares
            % than are in the portfolio.
            flag = 0;
            msg = 'Not enough shares to sell.';
            portfolioMod = portfolio;
            accountMod = account;
            return;
        end
    else
        % Error.  Can't sell a stock
        % that is not in the portfolio.
        flag = 0;
        msg = 'Stock not in portfolio.';
        portfolioMod = portfolio;
        accountMod = account;
        return;
    end

    % Record the date of the sell as the
    % last day of trading to date.
    portfolio.lastTradeDay_year = timeStamp(1);
    portfolio.lastTradeDay_month = timeStamp(2);
    portfolio.lastTradeDay_day = timeStamp(3);

    % Record information about the transaction.

    % Find the next empty row in the
    % transaction history list.
    nextTransNum = (size(portfolio.transactions,1) + 1);
    % Add the transaction information
    % to the list.
    portfolio.transactions{nextTransNum,1} = 'SELL';
    portfolio.transactions{nextTransNum,2} = timeStamp(1);
    portfolio.transactions{nextTransNum,3} = timeStamp(2);
    portfolio.transactions{nextTransNum,4} = timeStamp(3);
    portfolio.transactions{nextTransNum,5} = timeStamp(4);
    portfolio.transactions{nextTransNum,6} = timeStamp(5);
    portfolio.transactions{nextTransNum,7} = timeStamp(6);
    portfolio.transactions{nextTransNum,8} = symbol;
```

```
    [flag,tempStock] = getStockData_exchange(exchange,symbol);
    if(flag == 0)
        % Error.  Stock not found in exchange.
        flag = 0;
        msg = 'Stock not in exchange.';
        portfolioMod = portfolio;
        accountMod = account;
        return;
    else
        portfolio.transactions{nextTransNum,9} ...
            = tempStock.currentPrice;
    end
    portfolio.transactions{nextTransNum,10} = numShares;
    portfolio.transactions{nextTransNum,11} ...
        = (tempStock.currentPrice * numShares);

    % Update the portfolio with newly
    % calculated values.  a, b, and c are
    % dummies, just care about getting the
    % updated portfolio back from the function.
    [a,b,c,portfolio] = calcInvestment(portfolio,exchange);

    % Update the investment account
    % by adding/subtracting from
    % the balance.
    nextIndex = (length(account.year) + 1);
    account.year(nextIndex) = timeStamp(1);
    account.month(nextIndex) = timeStamp(2);
    account.day(nextIndex) = timeStamp(3);
    account.balance(nextIndex) =...
        (account.balance(nextIndex-1) ...
        + (tempStock.currentPrice * numShares) ...
        - commission);

    % Make sure to return the newly
    % modified portfolio struct
    % and account struct.
    portfolioMod = portfolio;
    accountMod = account;

    flag = 1;
    msg = 'Success!';
    return;

end
```

```
function [stockMod] = simStock_ExxonMobil01( ...
                        stockStruct, ...
                        currentDay)

    % This function pulls actual
    % stock price history data for
    % GIS (General Mills) from
    % Yahoo! Finance via the
    % SQQ script.  It reads the data
    % from a CSV file that has been
    % created beforehand.

    % The only that is changed in this
    % function is the closing price.

    % All values in the stock data struct
    % that are not changed will be set
    % to "-1" in their corresponding
    % matrices, denoting an empty value.

    % Pull current day price from
    % external CSV file for GIS.
    dataAll = csvread('./Historical_Stock_Data/XOM.csv');
    % Only need close price data.
    dataClose = dataAll(:,4);

    % Calculate new price.
    if((currentDay(3) <= length(dataClose)) ...
        && (currentDay(3) >= 1))
        newPrice = dataClose(currentDay(3));
    else
        % dataClose array is out
        % of bounds.  There is no
        % more historical stock
        % data to feed in.
        newPrice = 0;
    end

    % Update the stock data.
    newDataIndex = (length(stockStruct.year) + 1);
    stockStruct.currentPrice = newPrice;
    stockStruct.high(newDataIndex) = -1;
    stockStruct.low(newDataIndex) = -1;
    stockStruct.close(newDataIndex) = newPrice;
    stockStruct.volume(newDataIndex) = -1;

    % Return updated stock struct.
    stockMod = stockStruct;

    return;
```

```
end
```

```
function [stockMod] = simStock_GeneralMills01( ...
                        stockStruct, ...
                        currentDay)

    % This function pulls actual
    % stock price history data for
    % GIS (General Mills) from
    % Yahoo! Finance via the
    % SQQ script.  It reads the data
    % from a CSV file that has been
    % created beforehand.

    % The only that is changed in this
    % function is the closing price.

    % All values in the stock data struct
    % that are not changed will be set
    % to "-1" in their corresponding
    % matrices, denoting an empty value.

    % Pull current day price from
    % external CSV file for GIS.
    dataAll = csvread('./Historical_Stock_Data/GIS.csv');
    % Only need close price data.
    dataClose = dataAll(:,4);

    % Calculate new price.
    if((currentDay(3) <= length(dataClose)) ...
        && (currentDay(3) >= 1))
        newPrice = dataClose(currentDay(3));
    else
        % dataClose array is out
        % of bounds.  There is no
        % more historical stock
        % data to feed in.
        newPrice = 0;
    end

    % Update the stock data.
    newDataIndex = (length(stockStruct.year) + 1);
    stockStruct.currentPrice = newPrice;
    stockStruct.high(newDataIndex) = -1;
    stockStruct.low(newDataIndex) = -1;
    stockStruct.close(newDataIndex) = newPrice;
    stockStruct.volume(newDataIndex) = -1;

    % Return updated stock struct.
    stockMod = stockStruct;

    return;
```

```
end
```

```
function [stockMod] = simStock_Rand01(...
                        stockStruct, ...
                        currentDay)

    % This function will simulate the
    % random behavior of a stock price.

    % In this particular function, there
    % will be a 50% chance of the price
    % going either up or down.  The amount
    % by which it goes up or down will be
    % determined by a Gaussian (bell-curve)
    % distribution, causing larger
    % changes to be less likely.

    % The trading volume will be held
    % fixed at its initial value.

    % The high and low prices for the day
    % will also be held fixed in this
    % particular function.

    % The only that is changed in this
    % function is the closing price.

    % All values in the stock data struct
    % that are not changed will be set
    % to "-1" in their corresponding
    % matrices, denoting an empty value.

    % Set parameters by which to determine
    % the new stock price.  These are the
    % limits of the likely percent change
    % that a stock might see in one day of
    % trading.
    percChangeUpperLimit = 0.5;
    percChangeLowerLimit = 0.001;

    % Determine whether the stock price
    % will go up or down.
    if(rand(1,1) > 0.5)
        movDir = 1;
    else
        movDir = -1;
    end

    % Determine how much the stock
    % price will change.  Choose a
    % percentage change based on a
    % random Gaussian distribution and 2
    % limiting thresholds.  Modify
```

```
% The Gaussian distribution so that
% the 0-sigma (mean) value
% corresponds to the lower
% threshold and the 3-sigma
% value corresponds to the
% upper threshold.  Also, only
% Accept positive values from
% the distribution.

% First, check to see that the
% current price of the stock
% is not zero or negative.  If
% it is, then assume the company
% is not in business and return
% the new price as zero.
if(stockStruct.currentPrice <= 0)
    newPrice = 0;
else
    % Get a Gaussian distributed
    % random number.  Mean is at
    % 0 and standard deviation is 1.
    gaussVal = randn(1,1);
    % Shift random value so that the
    % distribution is now centered
    % at the percChangeLowerLimit.
    gaussVal = gaussVal + percChangeLowerLimit;
    % Stretch/condense the distribution
    % so that the 3-sigma value
    % corresponds to the upper percent
    % change limit.  The current 3-sigma
    % value is 3.
    gaussVal = (gaussVal*(0.5/3));
    % Take the absolute value of the
    % Gaussian random number.
    % This will be the percent change
    % of the stock price on the given day.
    % THE PERCENT CHANGE WILL BE A PERCENT
    % OF THE ORIGINAL PRICE!  OTHERWISE
    % THE STOCK PRICE WILL ALWAYS
    % TEND TO ZERO.  The original price
    % is given by "stockStruct.close(1)".
    percChange = abs(gaussVal);
    priceChange = (movDir ...
        * stockStruct.close(1) ...
        * percChange);
    newPrice = stockStruct.currentPrice ...
        + priceChange;
end

% If the new price is less than or
% equal to zero, then assume the
```

```
    % company has gone out of business
    % and the stock is now worthless.
    if(newPrice <= 0)
        newPrice = 0;
    end

    % Update the stock data.
    newDataIndex = (length(stockStruct.year) + 1);
    stockStruct.currentPrice = newPrice;
    stockStruct.high(newDataIndex) = -1;
    stockStruct.low(newDataIndex) = -1;
    stockStruct.close(newDataIndex) = newPrice;
    stockStruct.volume(newDataIndex) = -1;

    % Return updated stock struct.
    stockMod = stockStruct;

    return;

end
```

```
function [stockMod] = simStock_Rand02(...
                        stockStruct, ...
                        currentDay)

    % This function will simulate the
    % random behavior of a stock price.

    % In this particular function, there
    % will be a 50% chance of the price
    % going either up or down.  The amount
    % by which it goes up or down will be
    % determined by a Gaussian (bell-curve)
    % distribution, causing larger
    % changes to be less likely.

    % The trading volume will be held
    % fixed at its initial value.

    % The high and low prices for the day
    % will also be held fixed in this
    % particular function.

    % The only that is changed in this
    % function is the closing price.

    % All values in the stock data struct
    % that are not changed will be set
    % to "-1" in their corresponding
    % matrices, denoting an empty value.

    % Set parameters by which to determine
    % the new stock price.  These are the
    % limits of the likely percent change
    % that a stock might see in one day of
    % trading.
    % 0.5 = 50%.
    percChangeUpperLimit = 0.005;
    percChangeLowerLimit = 0.0001;

    % Determine whether the stock price
    % will go up or down.
    if(rand(1,1) > 0.5)
        movDir = 1;
    else
        movDir = -1;
    end

    % Determine how much the stock
    % price will change.  Choose a
    % percentage change based on a
    % random Gaussian distribution and 2
```

```
% limiting thresholds.  Modify
% The Gaussian distribution so that
% the 0-sigma (mean) value
% corresponds to the lower
% threshold and the 3-sigma
% value corresponds to the
% upper threshold.  Also, only
% Accept positive values from
% the distribution.

% First, check to see that the
% current price of the stock
% is not zero or negative.  If
% it is, then assume the company
% is not in business and return
% the new price as zero.
if(stockStruct.currentPrice <= 0)
    newPrice = 0;
else
    % Get a Gaussian distributed
    % random number.  Mean is at
    % 0 and standard deviation is 1.
    gaussVal = randn(1,1);
    % Shift random value so that the
    % distribution is now centered
    % at the percChangeLowerLimit.
    gaussVal = gaussVal + percChangeLowerLimit;
    % Stretch/condense the distribution
    % so that the 5-sigma value
    % corresponds to the upper percent
    % change limit.  The current 5-sigma
    % value is 5.
    gaussVal = (gaussVal*(0.5/5));
    % Take the absolute value of the
    % Gaussian random number.
    % This will be the percent change
    % of the stock price on the given day.
    % THE PERCENT CHANGE WILL BE A PERCENT
    % OF THE ORIGINAL PRICE!  OTHERWISE
    % THE STOCK PRICE WILL ALWAYS
    % TEND TO ZERO.  The original price
    % is given by "stockStruct.close(1)".
    percChange = abs(gaussVal);
    priceChange = (movDir ...
        * stockStruct.close(1) ...
        * percChange);
    newPrice = stockStruct.currentPrice ...
        + priceChange;
end

% If the new price is less than or
```

```
    % equal to zero, then assume the
    % company has gone out of business
    % and the stock is now worthless.
    if(newPrice <= 0)
        newPrice = 0;
    end

    % Update the stock data.
    newDataIndex = (length(stockStruct.year) + 1);
    stockStruct.currentPrice = newPrice;
    stockStruct.high(newDataIndex) = -1;
    stockStruct.low(newDataIndex) = -1;
    stockStruct.close(newDataIndex) = newPrice;
    stockStruct.volume(newDataIndex) = -1;

    % Return updated stock struct.
    stockMod = stockStruct;

    return;

end
```

```
function [stockMod] = simStock_Sine01(...
                        stockStruct, ...
                        currentDay)

    % This function will simulate the
    % behavior of a stock price as
    % a perfect sinewave.  It will
    % calculate the next value by
    % pulling it from a pre-set
    % array of values based on the
    % current day.  A separate
    % function will generate the array
    % of values for as many days as
    % are required.

    % The only that is changed in this
    % function is the closing price.

    % All values in the stock data struct
    % that are not changed will be set
    % to "-1" in their corresponding
    % matrices, denoting an empty value.

    % Set parameters by which to determine
    % the new stock price.

    % Amplitude ($).
    A = (0.10*stockStruct.close(1));
    % Period (days).
    T = 14;
    % Zero offset ($).
    % This will also be
    % the initial value.
    k = stockStruct.close(1);

    % Calculate the new price.
    secPerDay = 24*60*60;
    newPrice = (A*sin(2*pi*(1/(T*secPerDay)) ...
        *(currentDay(3)*secPerDay)) + k);

    % Update the stock data.
    newDataIndex = (length(stockStruct.year) + 1);
    stockStruct.currentPrice = newPrice;
    stockStruct.high(newDataIndex) = -1;
    stockStruct.low(newDataIndex) = -1;
    stockStruct.close(newDataIndex) = newPrice;
    stockStruct.volume(newDataIndex) = -1;

    % Return updated stock struct.
    stockMod = stockStruct;
```

```
    return;

end
```

```
function [exchangeMod] = updateExchange( ...
                         exchange, ...
                         currentTime)
    % This function will update the
    % given stock exchange based on
    % simulated market activity.

    for i = (1:length(exchange))
        % Simulate price activity for
        % each stock.
        [newStockData] = simStock_GeneralMills01( ...
            exchange(i),currentTime);
        % Update the market activity
        % of each stock in the exchange.
        nextEntry = (length(exchange(i).year) + 1);
        exchange(i).currentPrice = newStockData.currentPrice;
        exchange(i).year(nextEntry) = currentTime(1);
        exchange(i).month(nextEntry) = currentTime(2);
        exchange(i).day(nextEntry) = currentTime(3);
        exchange(i).high(nextEntry) = newStockData.high(end);
        exchange(i).low(nextEntry) = newStockData.low(end);
        exchange(i).close(nextEntry) = newStockData.close(end);
        exchange(i).volume(nextEntry) = newStockData.volume(end);
    end

    % Make sure to return the
    % modified excange so the
    % data is not lost.
    exchangeMod = exchange;

    return;

end
```

```
function [performanceMod] = updatePerformance( ...
                            performance, ...
                            portfolio, ...
                            dateStamp)

    % This function will update the
    % performance struct with the
    % data from the given portfolio
    % on the given date.

    % Determine the next element
    % for data.
    nextIndex = (length(performance.year) + 1);
    % Update data.
    performance.year(nextIndex) = dateStamp(1);
    performance.month(nextIndex) = dateStamp(2);
    performance.day(nextIndex) = dateStamp(3);
    performance.totalInvestment(nextIndex) = ...
        portfolio.totalInvestment;
    performance.totalRevenue(nextIndex) = ...
        portfolio.totalRevenue;
    performance.totalValue(nextIndex) = ...
        portfolio.totalValue;

    % Make sure to return the updated
    % performance struct so the data
    % is not lost.
    performanceMod = performance;

    return;

end
```

```
function [portfolioMod,accountMod] = updatePortfolio( ...
                        exchange, ...
                        portfolio, ...
                        account, ...
                        currentTime, ...
                        tradeCommission, ...
                        minTransProfit, ...
                        avgWindow)
    % This function will update the
    % given portfolio based on
    % simulated market activity.

    % In the end, after all
    % calculations have been done,
    % there will be an action determined
    % for each stock in the exchange.
    % The action will be in the following
    % form:
    %   1. Stock symbol
    %   2. Number of shares to act on
    %   3. Operation (buy/sell)
    % If the number of shares is zero,
    % this means "do nothing."

    % After it is determined what actions
    % will be taken on each stock in the
    % exchange, the changes will be made
    % using the "buy" and "sell" functions,
    % and the transactions will all be
    % recorded to the transaction list
    % by those same functions.

    % Loop through all stocks in the
    % exchange and decide how to
    % transact for each stock.
    for i = (1:length(exchange))
        % If the stock price is zero,
        % don't do anything.  Assume the
        % company has gone out of
        % business.  Skip ahead to
        % the next iteration of the loop.
        if(exchange(i).currentPrice == 0)
            continue;
        end
        % Retrieve the information
        % about the current stock from
        % the portfolio.
        [flag,currentStockPortfolioData] = ...
            getStockData_portfolio( ...
            portfolio,exchange(i).symbol);
        % Check the flag to make
```

```
% sure the function executed
% successfully.
if(flag == 0)
    % Stock not found in portfolio.
    % Handle error.
    fprintf('Stock not found in portfolio!\n');
    portfolioMod = portfolio;
    accountMod = account;
    return;
end

% The stock was found in the
% portfolio.  Continue with
% normal buy/sell decision
% making.

% Calculate the buy/sell
% threshold for the current stock.
[flag,thresh] = calcTransThresh_01( ...
    exchange,portfolio, ...
    exchange(i).symbol);     %,...
    %avgWindow);
% Check the flag to make
% sure the function executed
% successfully.
if(flag == 0)
    % Handle error.
    fprintf('Error calculating transaction threshold!\n');
    portfolioMod = portfolio;
    accountMod = account;
    return;
end
% Determine the stock price
% when the last transaction
% was executed.
lastTransPrice =...
    currentStockPortfolioData.transactions{end,9};
% Use the threshold ratio to
% determine how many shares of
% each stock to buy or sell.
% Account for the fixed
% trade commission from the
% broker.  The decision to
% buy or sell is based on the
% price change since the last
% transaction for this stock.

% Calculate price change.
priceChange = (exchange(i).currentPrice ...
    - lastTransPrice);
% Decide whether to buy or sell.
```

```
if(priceChange > 0)
    % Sell.
    transType = 'SELL';
elseif(priceChange < 0)
    % Buy.
    transType = 'BUY';
else
    % Do nothing.
    transType = 'NONE';
end
% Calculate how many shares
% to transact based only on the
% price change and threshold
% slope.  Round this number
% down to the nearest integer
% so the number of shares
% transacted is a whole number.
sharesToTrans = (abs(priceChange) * thresh);
sharesToTrans = floor(sharesToTrans);

% See if the proposed transaction
% will produce a portfolio
% value increase to cover the
% commission plus the desired
% profit per sale, set in the
% "parameters" script.  Pretend
% that either a buy or a sell is
% a sell.  This way, we can
% intuitively decide whether to
% transact based on profit, and
% then just mirror that decision
% and apply it to a buy as well.

% Calculate the gain/loss
% magnitude for the proposed
% transaction.
sharesOwn = currentStockPortfolioData.numShares;
gainLossMag = abs((sharesOwn ...
    * exchange(i).currentPrice) ...
    - (sharesOwn * lastTransPrice));
% If the magnitude of the
% gain warrants a sale,
% then execute the transaction.
% Since we are using absolute
% value, the decisions to buy
% or sell are both based on the
% magnitude of the gain, even
% if it is negative.  Since it
% doesn't make sense to calculate
% the buy threshold based on how
% much will be lost from the buy,
```

```
        % we simply mirror the sell
        % threshold and use it for buying
        % as well.
        if(gainLossMag >= (tradeCommission ...
                        + minTransProfit))
            % Yes, go ahead with
            % transaction.
            if(strcmp(transType,'SELL'))
                % Attempt to execute sale.
                [flag,msg,portfolio,account] = ...
                    sell(portfolio, ...
                    exchange, ...
                    account, ...
                    exchange(i).symbol, ...
                    sharesToTrans, ...
                    tradeCommission, ...
                    currentTime);
                if(flag == 1)
                    % Success!
                elseif(flag == 0)
                    % Try selling again
                    % with all shares owned.
                    % The sell function
                    % will terminate if
                    % it attempts to sell
                    % more shares than are
                    % owned.
                    [flag,msg,portfolio,account] = ...
                        sell(portfolio, ...
                        exchange, ...
                        account, ...
                        exchange(i).symbol, ...
                        sharesOwn, ...
                        tradeCommission, ...
                        currentTime);
                    if(flag == 1)
                        % Success!
                    elseif(flag == 0)
                        % Can't sell.
                        % Do nothing.
                    else
                        % Handle error.
                    end
                else
                    % Handle error.
                end
            elseif(strcmp(transType,'BUY'))
                % Attempt to execute buy.
                [flag,msg,portfolio,account] = ...
                    buy(portfolio, ...
                    exchange, ...
```

```
                    account, ...
                    exchange(i).symbol, ...
                    sharesToTrans, ...
                    tradeCommission, ...
                    currentTime);
                if(flag == 1)
                    % Success!
                elseif(flag == 0)
                    % Failed to buy.
                    % This should never
                    % happen unless the
                    % stock is not in the
                    % exchange, however that
                    % wouldn't be possible
                    % if control got to this
                    % point.
                    % Handle error.
                    fprintf('Failed to buy stock!\n');
                    portfolioMod = portfolio;
                    accountMod = account;
                    return;
                else
                    % Handle error.
                end
            elseif(strcmp(transType,'NONE'))
                % Do Nothing.
            else
                % Handle error.
            end
        else
            % No, don't go ahead
            % with transaction.
        end
    end

    % Make sure to return the
    % modified portfolio so the
    % data is not lost.
    portfolioMod = portfolio;
    accountMod = account;

    return;

end
```