Stock Smarts in a Market of Madness

New Mexico Supercomputing Challenge

Albuquerque Academy

Team 7

Team Members:

Jack Ingalls

Chad Bustard

Nathan Thai

Sponsoring Teacher and Mentor:

Jim Mims

Table of Contents

An appendix with a * next to it means that the class is courtesy of Didier H. Besset and is from his book, Object-oriented implementation of numerical methods an introduction with Java and Smalltalk

Executive Summary

We wrote a program that implements four models for predicting the stock market. The first model is the standard buy and hold method used by most investors who feel that they shouldn't try to predict the complicated workings of the stock market and, therefore, just buy a set amount of stock and sell everything after a certain period of time. The second and third models use the concept of moving averages (both simple and weighted) to see whether the stock is trending up or down, indicating whether to buy or sell. The third model is our own, which uses the concept that you keep a constant amount of money in a stock regardless of whatever else the stock is doing. Of these three models, we have found that the moving averages and the constant investment models gave the most profit when used on past stock data from various stocks. The buy and hold method worked the worst overall, sometimes losing money. We are currently working on a Fourier model to closely model the cyclic nature of a stock. We should have this completed by the Expo in April. However, we currently have evidence that suggests our constant investment method is better, simpler, and easier to calculate than many of the others.

Problem Statement

The stock market has gone through times of prosperity and times of loss. From the boom of the 1920's, to the Great Depression of the 1930's, to the moderate climb of the 1950's to 1980's, to the crash of 1987, to the tech boom of the 1990's, to the slump in the new millennium, and to the crisis today, the Stock Market has been a perplexing engine of growth, profit, and loss.

Throughout time, people have been trying to figure out how to predict the stock market. Although it usually averages adequate returns of around 10% a year, greed causes people to yearn for more money, and those people will often jump on a stock that is rising at 5% a month. However, history repeatedly shows that this is not the way to make money in the stock market. Fear quickly sets in and causes these same stocks to plummet.

Many people know little about how they should and should not invest. Some people try to build off of early successes in the market by investing too heavily, ultimately causing them to lose money. This causes good people to have to work many more years just because of improper investing. On the other side of the spectrum, some people do not invest at all, or invest lightly during their early years, only to realize later that they could have had a lot more if they had just risked a little bit each year. Also, so many people do not understand how the stock market works, but they try to chase profits and end up on the wrong side of trends. People need to know how to invest so that they can maximize their money while still being secure.

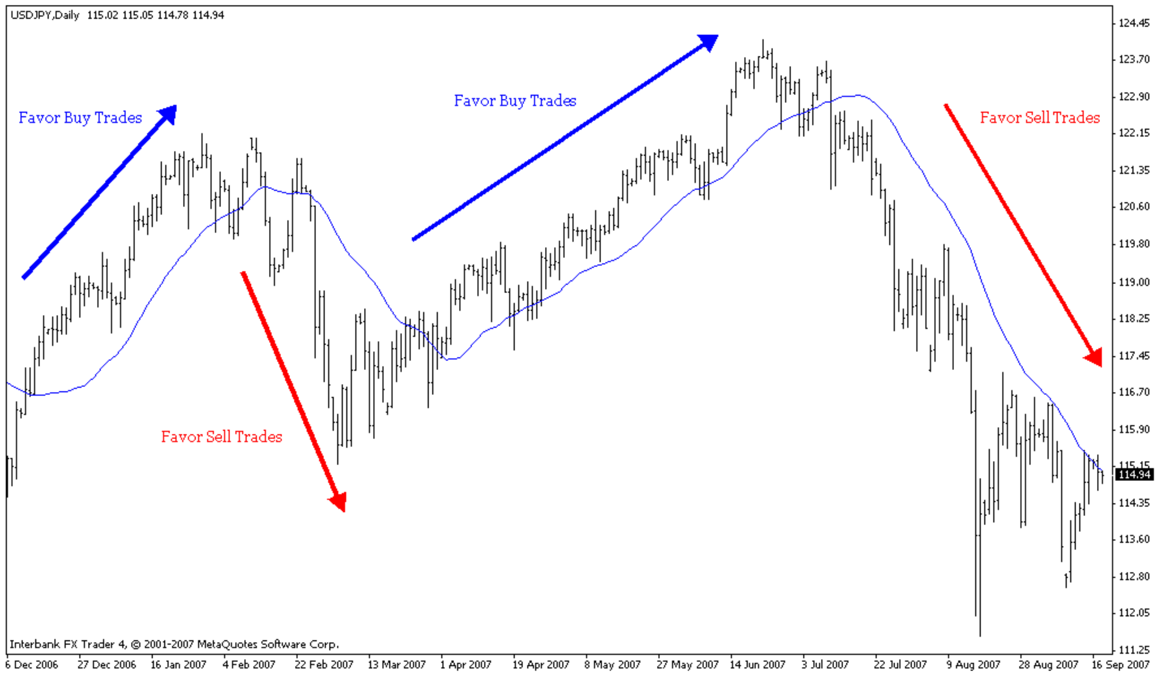
Our Approaches

Buy and Hold

The buy and hold method is used by most people who invest in the stock market. It consists of simply buying a number of shares, waiting for a period of time, then selling. Most people use this method because they don't know what else to do to make money in the stock market. This is the simplest method because it involves little strategy. You do not make any attempt to sell or buy when losing or making money. This method can essentially be called the control or baseline for our project

Moving Averages:

For calculating a moving average, you take a certain set of points, for example, five consecutive days of closing stock prices, and calculate their average. It is called a moving average because the average moves with the stock and the moving average can easily be graphed with the actual stock price. Moving averages are lagging indicators, meaning that the value they give is always past the current trend of the market. This means that if the stock price were rising, the moving average would most likely be below the actual price. If the stock price were falling, the moving average would most likely be above the actual price. This lagging nature of the moving average can be used to decide whether to buy or sell. Thus, when the moving average graph crosses the actual stock price graph, it indicates a buy (if the moving average is then below) or a sell (if the moving average is then above). The graph below indicates this principle. If the stock is not trending up or down, though, this can give misleading results and possibly lead to bad investments.

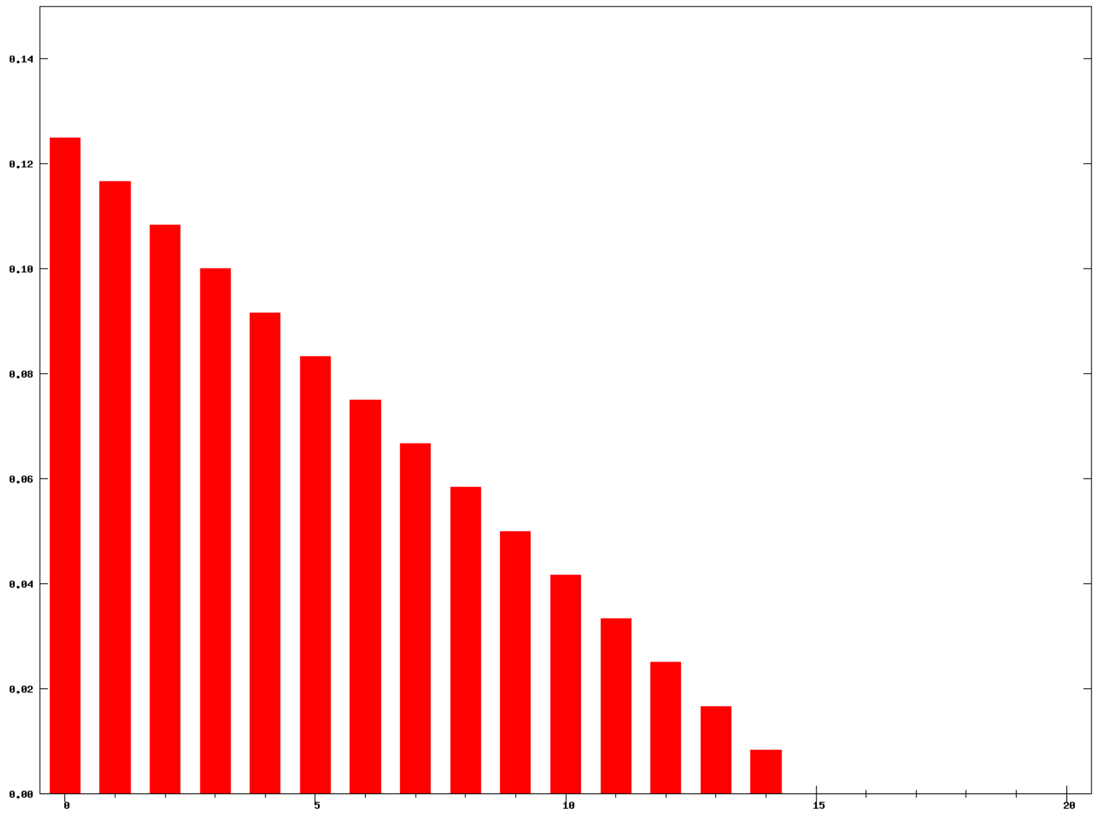


Graph of an example stock and 30 day moving average (in blue).

http://smarttradingforprofits.com/wp-content/uploads/2007/09/usdjpy-30-day-sma.gif

Weighted Moving Averages:

The weighted moving average is similar to the moving average model; however, it weights the most recent data values more. For example, if you calculate the weighted moving average for the past ten days, the value for the most distant day will be multiplied by one, the second day by two, the third day by three, and so on. The value for the most recent day will be multiplied by ten. This would make the model more sensitive to recent changes in the market, and, supposedly, more accurate for prediction. The chart below indicates how values are weighted.



Shows the weights of the weighted moving average (x-axis in days ago)

http://en.wikipedia.org/wiki/File:Weighted_moving_average_weights_N%3D15.png

Constant Investment:

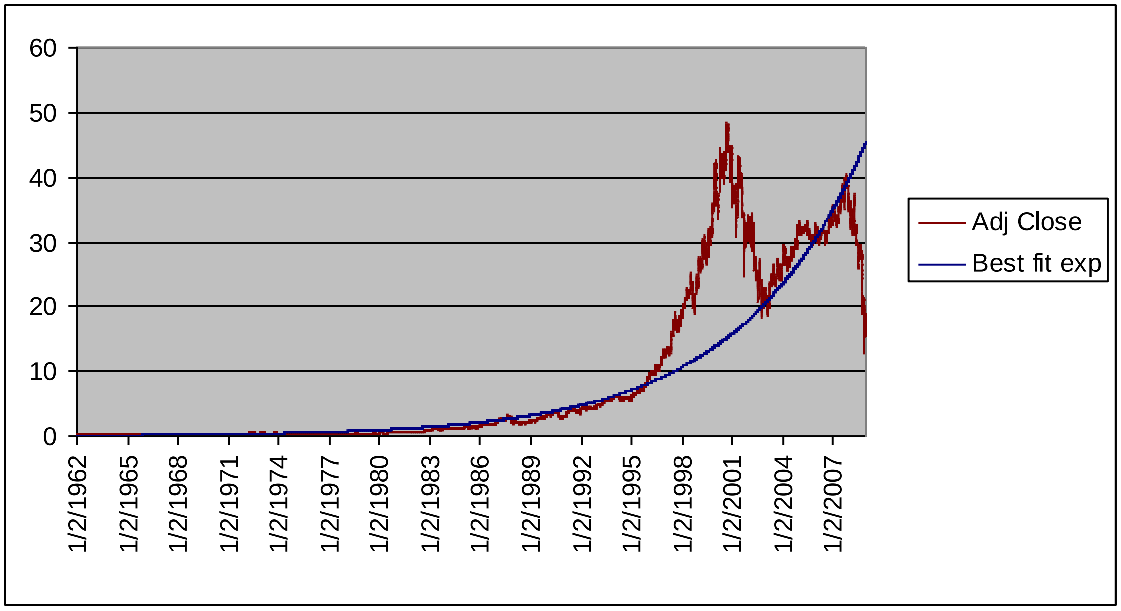
This method follows the rule that our investment in a stock will remain constant. Therefore, if we want to invest $15000 in one stock and it is at $10 a share, we will buy 1500 shares. If it goes up 50% to $15 a share, our investment will then be $22,500, so we will sell 500 shares and keep 1000 shares, pocket $7,500 of profit, and have $15,000 invested in the company. If the stock drops 33% and returns to $10 a share, then we will buy back 500 shares to have 1500 shares and spend $5,000. Thus, the overall price of the stock will remain constant, and we make a profit of $2,500. This method works from the idea that stocks are impossible to predict, but usually fluctuate and will generally trend upwards, so it's best to have a method that makes money from fluctuations. This method also comes from the concept that buying low and selling high will give one the most profit. In addition, one should note that if the stock went down 33% first and then went up 50%, the stock overall would not have changed in price but we still would have made a 16% profit ($2,500).

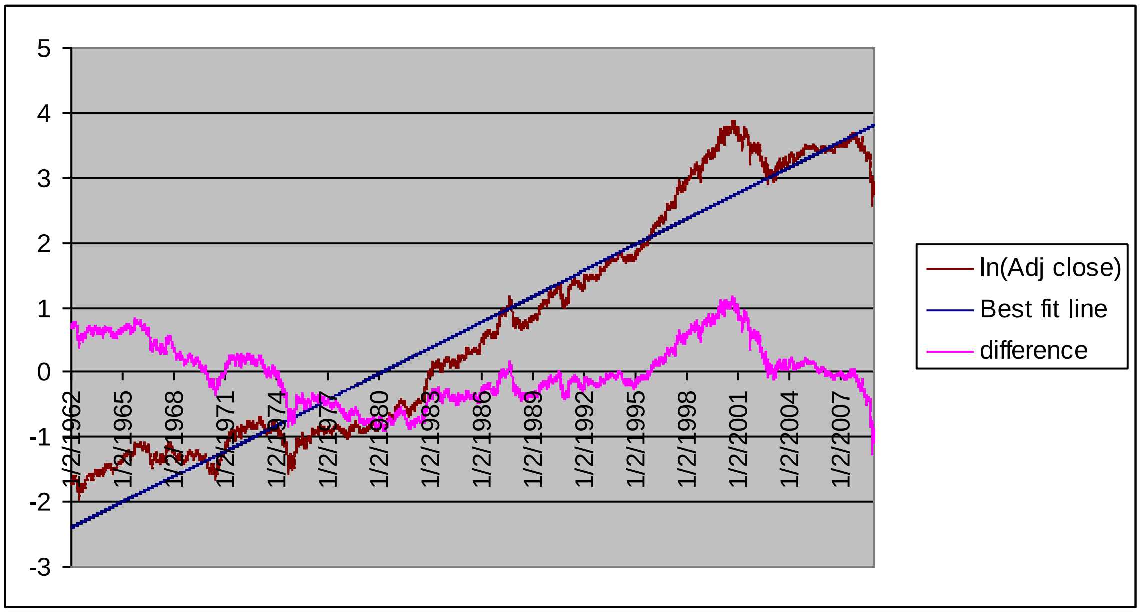
Fourier Transform Model:

For this model, we are assuming that stocks tend to follow exponential growth with fluctuations. Therefore, we take the natural log of the price function to obtain a graph that is linear with fluctuations. After finding a best-fit line, we find the residual. The residuals plot, which graphed the actual value of the natural log of the price minus the value of the best-fit line for each day, showed a pattern resembling a sum of sines and cosines centered about the "day" axis. The form of a Fourier series
is:

$$(S_N f)(x) = \frac{a_0}{2} + \sum_{n=1}^{N} \left[a_n \cos(nx) + b_n \sin(nx)\right], \quad N \geq 0.$$

where N is a finite but very large integer. However, we need to have the flexibility to choose the best frequency that fits our given data because our period will probably not be known. To do this, we expect to have to find the half-period, L, of the data. To find L, we will use a separate method that takes the data points for the residuals and finds where they switch from positive to negative. If the switches occur too close too each other, they need to be treated like one overall change. We then find the distance in days between sign changes for the entire set of data, average them, and use this value as an estimated half period, L_0. We will then have further challenges finding the best period, but our estimation will be in the ballpark. We have included our code for the Fourier class and the classes it uses; however, it is currently incomplete.





Math Model

Our model is centered around the class *StockMarketDriver*. This initiates an *ArrayList* that contains all stocks. Thus, the *ArrayList* “stocks” contains instances of the *Stock* class. The driver then initiates all of the necessary computations. The *Stock* class itself contains an A*rrayList* “entries” which are instances of the S*tockDay* class. The *StockDay* class contains information of one day of a stock. Therefore, it contains data like closing price and volume, and it also has a method for determining which place to put the data according to where it was in a spreadsheet (more detail will be given in *ReadExcel*).

As *Stock* is initiated, the constructor takes a String argument, which says what stock this stock will be (for example, JAVA for Sun Microsystems). It then stores this information for later use. Then the driver will call the retrieveData method from each stock. The retrieveData finds calls the read method in the *ReadExcel* class and passes the tag. The *ReadExcel* class then imports all of the data from the given excel spreadsheet, which is located in a references folder in our project. These spreadsheets are the same as the ones that can be found on Yahoo Finance. After retrieving data from a row, it allows the *StockDay* class to organize the data, and continues row by row until the *ArrayList* is filled and the program reaches the end of the spreadsheet.

After that, the driver calls for output from that stock. The stock then outputs basic information and calls other methods to calculate what the ideal method for buying and selling stocks is. In order to make it fair, we set aside the last 2,500 days as testing days, and so our program represents a person ready to enter the stock market around 10 years ago. After calculating which moving average and weighted average worked best in the past, our imaginary person then uses that method for the next 10 years (up until 03/30/09). Then, the program will output how well the methods worked and see if one was the best at making money.

The driver then cycles through the rest of the stocks and does the same thing.

During debugging, we found that our program was very slow. Our first program took around 6 minutes to run, included 1 stock, and only had to find the best moving average. We found that we were calculating the moving average every time from scratch instead of using the previous moving average. Our method added up any given period worth of prices. The period could be as long as the number of days we have for data. However, if you use the previous moving average, you then add on the value for the next day and subtract off the oldest day (and take this change and divide by the period) to calculate the next average. This reduced our program from $O(n^3)$ to $O(n^2)$, a major improvement. Similarly, the weighted average can be computed each time for a big O notation of $O(n^3)$, or it can subtract off the moving average and add the most recent day (this difference needs to be divided by the period + 1) and receive a big O notation of $O(n^2)$. The buy and hold method is O(1) and the constant investment method is O(n), so our program turned out, overall, to be on the order of the number of stocks times the number of days in the stock squared. Our final program analyzed 20 stocks with all of the mentioned methods (accept for fourier) and ran in 2 minutes, 1 second.

The Fourier method is unfortunately incomplete, but it uses all of the borrowed classes to find the best fit line. All of the other methods use entirely original code except to use the *ReadExcel*, jlx.jar, and other basic Java classes.

Results

Here is the output of our program excluding the output from the Fourier model. It outputs the name of the stock, then the size of the data from the stock, then the best methods for the past, and then how those methods fare using “future” data (data where the method was determined using data before it, so it is like on 04/21/99 or so, we looked at past data, decided on a method, and then used that method for the next 10 years). It does this same process over the 20 stocks that we have.

```
Data for BAC.
Size: 5760

For 1986-05-29 until 1999-04-21, these were the best results.
The best moving average was: 2.
It gave a profit of $2906469.4099999946.
Which gave a percent increase of 2906.4694099999947%.
The best weighted average was: 3.
It gave a profit of $3808780.160000008.
Which gave a percent increase of 3808.780160000008%.
The Constant Investment method gave a profit of $271661.250000002.
Which gave a percent increase of 271.661250000002%.
The 'buy and hold' method gave a profit of $717931.0344827586.
Which was a percent increase of 717.9310344827586%.

Using these strategies, the following results were achieved between 1999-04-22 and 2009-03-30.
The best moving average was: 2.
It now gives a profit of $304.3800000001065.
Which gives a percent increase of 0.3043800000001065%.
The best weighted average was: 3.
```

It now gives a profit of $16231.589999999924.

Which gives a percent increase of 16.231589999999922%.

The Constant Investment method now gives a profit of $-4388.269999999611.

Which gives a percent increase of -4.388269999999611%.

The 'buy and hold' method now gives a profit of $-74927.23492723492.

Which gives a percent increase of -74.92723492723492%.

Data for CMCSA.

Size: 5221

For 1988-07-07 until 1999-04-13, these were the best results.

The best moving average was: 2.

It gave a profit of $595965.1600000003.

Which gave a percent increase of 595.9651600000003%.

The best weighted average was: 2.

It gave a profit of $815614.1199999994.

Which gave a percent increase of 815.6141199999994%.

The Constant Investment method gave a profit of $314450.59000000136.

Which gave a percent increase of 314.45059000000134%.

The 'buy and hold' method gave a profit of $892000.0.

Which was a percent increase of 892.0%.

Using these strategies, the following results were achieved between 1999-04-14 and 2009-03-30.

The best moving average was: 2.

It now gives a profit of $-79280.65000000005.

Which gives a percent increase of -79.28065000000005%.

The best weighted average was: 2.

It now gives a profit of $-81313.09000000016.

Which gives a percent increase of -81.31309000000016%.

The Constant Investment method now gives a profit of $41641.430000000575.

Which gives a percent increase of 41.641430000000575%.

The 'buy and hold' method now gives a profit of $-37926.330150068214.

Which gives a percent increase of -37.92633015006821%.

Data for COST.

Size: 5732

For 1986-07-09 until 1999-04-21, these were the best results.

The best moving average was: 102.

It gave a profit of $416423.6400000001.

Which gave a percent increase of 416.4236400000001%.

The best weighted average was: 2487.

It gave a profit of $354752.86.

Which gave a percent increase of 354.75286%.

The Constant Investment method gave a profit of $225506.77000000083.

Which gave a percent increase of 225.50677000000084%.

The 'buy and hold' method gave a profit of $282234.9570200573.

Which was a percent increase of 282.2349570200573%.

Using these strategies, the following results were achieved between 1999-04-22 and 2009-03-30.

The best moving average was: 102.

It now gives a profit of $-41033.96000000007.

Which gives a percent increase of -41.03396000000007%.

The best weighted average was: 2487.

It now gives a profit of $0.0.

Which gives a percent increase of 0.0%.

The Constant Investment method now gives a profit of $75653.45000000001.

Which gives a percent increase of 75.65345%.

The 'buy and hold' method now gives a profit of $11823.017408123793.

Which gives a percent increase of 11.823017408123793%.

Data for DE.

Size: 6867

For 1982-01-04 until 1999-04-13, these were the best results.

The best moving average was: 2.

It gave a profit of $6597883.4100000085.

Which gave a percent increase of 6597.883410000009%.

The best weighted average was: 2.

It gave a profit of $4256102.260000017.

Which gave a percent increase of 4256.102260000017%.

The Constant Investment method gave a profit of $322559.6100000001.

Which gave a percent increase of 322.5596100000001%.

The 'buy and hold' method gave a profit of $986842.105263158.

Which was a percent increase of 986.842105263158%.

Using these strategies, the following results were achieved between 1999-04-14 and 2009-03-30.

The best moving average was: 2.

It now gives a profit of $-19265.780000000115.

Which gives a percent increase of -19.265780000000117%.

The best weighted average was: 2.

It now gives a profit of $-8389.300000000221.

Which gives a percent increase of -8.38930000000022%.

The Constant Investment method now gives a profit of $136731.5.

Which gives a percent increase of 136.7315%.

The 'buy and hold' method now gives a profit of $86421.17376294592.

Which gives a percent increase of 86.42117376294593%.

```
Data for DOW.

Size: 8135


For 1977-01-03 until 1999-04-20, these were the best results.

The best moving average was: 2.

It gave a profit of $4869710.449999994.

Which gave a percent increase of 4869.710449999993%.

The best weighted average was: 2.

It gave a profit of $3141672.889999997.

Which gave a percent increase of 3141.672889999997%.

The Constant Investment method gave a profit of $321872.1699999994.

Which gave a percent increase of 321.8721699999994%.

The 'buy and hold' method gave a profit of $1017948.717948718.

Which was a percent increase of 1017.948717948718%.


Using these strategies, the following results were achieved between 1999-04-21 and 2009-03-30.

The best moving average was: 2.

It now gives a profit of $-73434.81000000013.

Which gives a percent increase of -73.43481000000013%.

The best weighted average was: 2.

It now gives a profit of $-75236.58000000019.

Which gives a percent increase of -75.23658000000019%.

The Constant Investment method now gives a profit of $-45471.01999999996.

Which gives a percent increase of -45.47101999999996%.

The 'buy and hold' method now gives a profit of $-67741.93548387097.

Which gives a percent increase of -67.74193548387098%.
```

Data for EXC.

Size: 7379

For 1980-01-02 until 1999-04-21, these were the best results.

The best moving average was: 137.

It gave a profit of $1510732.4799999997.

Which gave a percent increase of 1510.7324799999997%.

The best weighted average was: 210.

It gave a profit of $1601952.8699999996.

Which gave a percent increase of 1601.9528699999996%.

The Constant Investment method gave a profit of $347436.7699999984.

Which gave a percent increase of 347.4367699999984%.

The 'buy and hold' method gave a profit of $1998809.5238095238.

Which was a percent increase of 1998.8095238095239%.

Using these strategies, the following results were achieved between 1999-04-22 and 2009-03-30.

The best moving average was: 137.

It now gives a profit of $184985.94999999984.

Which gives a percent increase of 184.98594999999983%.

The best weighted average was: 210.

It now gives a profit of $120889.71999999997.

Which gives a percent increase of 120.88971999999997%.

The Constant Investment method now gives a profit of $141717.57000000062.

Which gives a percent increase of 141.71757000000062%.

The 'buy and hold' method now gives a profit of $162346.04105571844.

Which gives a percent increase of 162.34604105571844%.

Data for F.

Size: 8136

For 1977-01-03 until 1999-04-21, these were the best results.

The best moving average was: 833.

It gave a profit of $6606235.61.

Which gave a percent increase of 6606.235610000001%.

The best weighted average was: 1257.

It gave a profit of $7112353.2.

Which gave a percent increase of 7112.3532000000005%.

The Constant Investment method gave a profit of $504889.7400000002.

Which gave a percent increase of 504.88974000000024%.

The 'buy and hold' method gave a profit of $5220000.0.

Which was a percent increase of 5220.0%.

Using these strategies, the following results were achieved between 1999-04-22 and 2009-03-30.

The best moving average was: 833.

It now gives a profit of $-41479.17999999997.

Which gives a percent increase of -41.47917999999997%.

The best weighted average was: 1257.

It now gives a profit of $-26730.89999999998.

Which gives a percent increase of -26.73089999999998%.

The Constant Investment method now gives a profit of $-107002.29999999992.

Which gives a percent increase of -107.00229999999992%.

The 'buy and hold' method now gives a profit of $-89624.06015037594.

Which gives a percent increase of -89.62406015037594%.

Data for GE.

Size: 11892

For 1962-01-02 until 1999-04-21, these were the best results.

The best moving average was: 3241.

It gave a profit of $9772397.18.

Which gave a percent increase of 9772.39718%.

The best weighted average was: 3438.

It gave a profit of $8342901.449999999.

Which gave a percent increase of 8342.90145%.

The Constant Investment method gave a profit of $652692.4999999901.

Which gave a percent increase of 652.6924999999901%.

The 'buy and hold' method gave a profit of $1.5805555555555554E7.

Which was a percent increase of 15805.555555555555%.

Using these strategies, the following results were achieved between 1999-04-22 and 2009-03-30.

The best moving average was: 3241.

It now gives a profit of $0.0.

Which gives a percent increase of 0.0%.

The best weighted average was: 3438.

It now gives a profit of $0.0.

Which gives a percent increase of 0.0%.

The Constant Investment method now gives a profit of $-45532.310000001075.

Which gives a percent increase of -45.532310000001075%.

The 'buy and hold' method now gives a profit of $-65218.91418563923.

Which gives a percent increase of -65.21891418563924%.

Data for GIS.

Size: 6503

For 1983-06-10 until 1999-04-13, these were the best results.

The best moving average was: 2.

It gave a profit of $1498612.8099999998.

Which gave a percent increase of 1498.6128099999999%.

The best weighted average was: 2.

It gave a profit of $2712642.869999999.

Which gave a percent increase of 2712.642869999999%.

The Constant Investment method gave a profit of $334167.19000000245.

Which gave a percent increase of 334.16719000000245%.

The 'buy and hold' method gave a profit of $1706134.9693251536.

Which was a percent increase of 1706.1349693251536%.

Using these strategies, the following results were achieved between 1999-04-14 and 2009-03-30.

The best moving average was: 2.

It now gives a profit of $-41857.24000000029.

Which gives a percent increase of -41.85724000000029%.

The best weighted average was: 2.

It now gives a profit of $-39628.45000000015.

Which gives a percent increase of -39.62845000000015%.

The Constant Investment method now gives a profit of $73575.62999999919.

Which gives a percent increase of 73.5756299999992%.

The 'buy and hold' method now gives a profit of $69854.68063534977.

Which gives a percent increase of 69.85468063534977%.

Data for HNZ.

Size: 6118

For 1984-12-17 until 1999-04-13, these were the best results.

The best moving average was: 750.

It gave a profit of $700987.16.

Which gave a percent increase of 700.98716%.

The best weighted average was: 871.

It gave a profit of $547270.2100000001.

Which gave a percent increase of 547.2702100000001%.

The Constant Investment method gave a profit of $328305.48999999894.

Which gave a percent increase of 328.30548999999894%.

The 'buy and hold' method gave a profit of $1614361.7021276595.

Which was a percent increase of 1614.3617021276596%.

Using these strategies, the following results were achieved between 1999-04-14 and 2009-03-30.

The best moving average was: 750.

It now gives a profit of $-11198.519999999917.

Which gives a percent increase of -11.198519999999917%.

The best weighted average was: 871.

It now gives a profit of $23668.159999999945.

Which gives a percent increase of 23.668159999999947%.

The Constant Investment method now gives a profit of $30346.82999999952.

Which gives a percent increase of 30.34682999999952%.

The 'buy and hold' method now gives a profit of $2978.591374495809.

Which gives a percent increase of 2.978591374495809%.

Warning: Property storage name for 5 is empty - setting to Root Entry

Data for JAVA.

Size: 5562

For 1987-03-11 until 1999-04-21, these were the best results.

The best moving average was: 2.

It gave a profit of $6577245.990000006.

Which gave a percent increase of 6577.245990000006%.

The best weighted average was: 2.

It gave a profit of $4130872.120000003.

Which gave a percent increase of 4130.872120000003%.

The Constant Investment method gave a profit of $505091.8499999996.

Which gave a percent increase of 505.0918499999996%.

The 'buy and hold' method gave a profit of $2935714.285714286.

Which was a percent increase of 2935.714285714286%.

Using these strategies, the following results were achieved between 1999-04-22 and 2009-03-30.

The best moving average was: 2.

It now gives a profit of $-85576.9199999998.

Which gives a percent increase of -85.57691999999979%.

The best weighted average was: 2.

It now gives a profit of $-86770.5899999998.

Which gives a percent increase of -86.7705899999998%.

The Constant Investment method now gives a profit of $-9211.659999999654.

Which gives a percent increase of -9.211659999999654%.

The 'buy and hold' method now gives a profit of $-88587.64186633039.

Which gives a percent increase of -88.58764186633039%.

Data for JNJ.

Size: 9905

For 1970-01-02 until 1999-04-21, these were the best results.

The best moving average was: 2.

It gave a profit of $1.3549482780000009E7.

Which gave a percent increase of 13549.48278000001%.

The best weighted average was: 2.

It gave a profit of $1.1843968679999994E7.

Which gave a percent increase of 11843.968679999994%.

The Constant Investment method gave a profit of $522218.55000000005.

Which gave a percent increase of 522.21855%.

The 'buy and hold' method gave a profit of $6950847.4576271195.

Which was a percent increase of 6950.847457627119%.

Using these strategies, the following results were achieved between 1999-04-22 and 2009-03-30.

The best moving average was: 2.

It now gives a profit of $-6535.830000000395.

Which gives a percent increase of -6.535830000000395%.

The best weighted average was: 2.

It now gives a profit of $-10534.6799999996.

Which gives a percent increase of -10.5346799999996%.

The Constant Investment method now gives a profit of $49221.73000000033.

Which gives a percent increase of 49.22173000000033%.

The 'buy and hold' method now gives a profit of $25259.924385633276.

Which gives a percent increase of 25.259924385633276%.

Data for LMT.

Size: 8136

For 1977-01-03 until 1999-04-21, these were the best results.

The best moving average was: 2.

It gave a profit of $2.9664248930000003E7.

Which gave a percent increase of 29664.248930000005%.

The best weighted average was: 2.

It gave a profit of $3.707456588000001E7.

Which gave a percent increase of 37074.56588000001%.

The Constant Investment method gave a profit of $556412.480000003.

Which gave a percent increase of 556.412480000003%.

The 'buy and hold' method gave a profit of $6598076.923076923.

Which was a percent increase of 6598.076923076923%.

Using these strategies, the following results were achieved between 1999-04-22 and 2009-03-30.

The best moving average was: 2.

It now gives a profit of $-50438.06999999978.

Which gives a percent increase of -50.43806999999978%.

The best weighted average was: 2.

It now gives a profit of $-47904.989999999794.

Which gives a percent increase of -47.90498999999979%.

The Constant Investment method now gives a profit of $118290.67999999906.

Which gives a percent increase of 118.29067999999906%.

The 'buy and hold' method now gives a profit of $99048.16844534178.

Which gives a percent increase of 99.04816844534179%.

Data for LUV.

Size: 7379

For 1980-01-02 until 1999-04-21, these were the best results.

The best moving average was: 2.

It gave a profit of $6330234.3199999975.

Which gave a percent increase of 6330.234319999998%.

The best weighted average was: 2.

It gave a profit of $1.0844908499999985E7.

Which gave a percent increase of 10844.908499999985%.

The Constant Investment method gave a profit of $598069.2199999983.

Which gave a percent increase of 598.0692199999984%.

The 'buy and hold' method gave a profit of $8416666.666666668.

Which was a percent increase of 8416.666666666668%.

Using these strategies, the following results were achieved between 1999-04-22 and 2009-03-30.

The best moving average was: 2.

It now gives a profit of $-91348.09000000011.

Which gives a percent increase of -91.34809000000011%.

The best weighted average was: 2.

It now gives a profit of $-91584.76.

Which gives a percent increase of -91.58475999999999%.

The Constant Investment method now gives a profit of $-10259.989999999845.

Which gives a percent increase of -10.259989999999846%.

The 'buy and hold' method now gives a profit of $-57623.62637362637.

Which gives a percent increase of -57.62362637362637%.

Data for MCD.

Size: 9905

For 1970-01-02 until 1999-04-21, these were the best results.

The best moving average was: 2.

It gave a profit of $1.510370042999998E7.

Which gave a percent increase of 15103.700429999979%.

The best weighted average was: 4.

It gave a profit of $1.441793227999999E7.

Which gave a percent increase of 14417.93227999999%.

The Constant Investment method gave a profit of $643836.5700000003.

Which gave a percent increase of 643.8365700000003%.

The 'buy and hold' method gave a profit of $1.5787500000000004E7.

Which was a percent increase of 15787.500000000004%.

Using these strategies, the following results were achieved between 1999-04-22 and 2009-03-30.

The best moving average was: 2.

It now gives a profit of $-15221.600000000326.

Which gives a percent increase of -15.221600000000326%.

The best weighted average was: 4.

It now gives a profit of $-36947.80999999993.

Which gives a percent increase of -36.94780999999993%.

The Constant Investment method now gives a profit of $77684.89999999941.

Which gives a percent increase of 77.68489999999942%.

The 'buy and hold' method now gives a profit of $43595.206391478045.

Which gives a percent increase of 43.595206391478044%.

Data for MOT.

Size: 8136

For 1977-01-03 until 1999-04-21, these were the best results.

The best moving average was: 3.

It gave a profit of $1.4119755579999998E7.

Which gave a percent increase of 14119.75558%.

The best weighted average was: 2.

It gave a profit of $1.4285798220000023E7.

Which gave a percent increase of 14285.798220000022%.

The Constant Investment method gave a profit of $450145.92999999865.

Which gave a percent increase of 450.14592999999866%.

The 'buy and hold' method gave a profit of $2303370.786516854.

Which was a percent increase of 2303.370786516854%.

Using these strategies, the following results were achieved between 1999-04-22 and 2009-03-30.

The best moving average was: 3.

It now gives a profit of $-52863.199999999975.

Which gives a percent increase of -52.86319999999998%.

The best weighted average was: 2.

It now gives a profit of $-19046.729999999763.

Which gives a percent increase of -19.046729999999762%.

The Constant Investment method now gives a profit of $-25476.820000000036.

Which gives a percent increase of -25.476820000000036%.

The 'buy and hold' method now gives a profit of $-80671.88219052002.

Which gives a percent increase of -80.67188219052002%.

Data for SLM.

Size: 5355

For 1988-01-05 until 1999-04-21, these were the best results.

The best moving average was: 2.

It gave a profit of $3195588.2400000007.

Which gave a percent increase of 3195.5882400000005%.

The best weighted average was: 2.

It gave a profit of $2959998.059999999.

Which gave a percent increase of 2959.998059999999%.

The Constant Investment method gave a profit of $236239.8200000011.

Which gave a percent increase of 236.23982000000112%.

The 'buy and hold' method gave a profit of $527722.7722772277.

Which was a percent increase of 527.7227722772277%.

Using these strategies, the following results were achieved between 1999-04-22 and 2009-03-30.

The best moving average was: 2.

It now gives a profit of $-51844.89000000053.

Which gives a percent increase of -51.84489000000053%.

The best weighted average was: 2.

It now gives a profit of $-52918.71999999985.

Which gives a percent increase of -52.91871999999985%.

The Constant Investment method now gives a profit of $39176.820000000065.

Which gives a percent increase of 39.17682000000006%.

The 'buy and hold' method now gives a profit of $-65255.5910543131.

Which gives a percent increase of -65.2555910543131%.

Data for SUN.

Size: 6555

For 1983-04-06 until 1999-04-21, these were the best results.

The best moving average was: 3.

It gave a profit of $455915.8099999997.

Which gave a percent increase of 455.9158099999997%.

The best weighted average was: 2.

It gave a profit of $458535.57999999914.

Which gave a percent increase of 458.53557999999913%.

The Constant Investment method gave a profit of $229755.84000000218.

Which gave a percent increase of 229.75584000000217%.

The 'buy and hold' method gave a profit of $402693.6026936027.

Which was a percent increase of 402.69360269360266%.

Using these strategies, the following results were achieved between 1999-04-22 and 2009-03-30.

The best moving average was: 3.

It now gives a profit of $-8680.079999999696.

Which gives a percent increase of -8.680079999999696%.

The best weighted average was: 2.

It now gives a profit of $-29738.55999999975.

Which gives a percent increase of -29.73855999999975%.

The Constant Investment method now gives a profit of $136827.3099999997.

Which gives a percent increase of 136.8273099999997%.

The 'buy and hold' method now gives a profit of $78638.81401617252.

Which gives a percent increase of 78.63881401617252%.

Data for WMT.

Size: 9233

For 1972-08-25 until 1999-04-21, these were the best results.

The best moving average was: 741.

It gave a profit of $1.1007733869000001E8.

Which gave a percent increase of 110077.33869000002%.

The best weighted average was: 1190.

It gave a profit of $7.867699391000001E7.

Which gave a percent increase of 78676.99391%.

The Constant Investment method gave a profit of $1229082.9100000006.

Which gave a percent increase of 1229.0829100000005%.

The 'buy and hold' method gave a profit of $8.78E7.

Which was a percent increase of 87800.0%.

Using these strategies, the following results were achieved between 1999-04-22 and 2009-03-30.

The best moving average was: 741.

It now gives a profit of $-57095.1800000001.

Which gives a percent increase of -57.0951800000001%.

The best weighted average was: 1190.

It now gives a profit of $-38160.00999999999.

Which gives a percent increase of -38.160009999999986%.

The Constant Investment method now gives a profit of $58289.76000000024.

Which gives a percent increase of 58.28976000000024%.

The 'buy and hold' method now gives a profit of $14690.892975847542.

Which gives a percent increase of 14.690892975847543%.

Data for XOM.

Size: 9905

For 1970-01-02 until 1999-04-21, these were the best results.

The best moving average was: 1266.

It gave a profit of $7802500.03.

Which gave a percent increase of 7802.50003%.

The best weighted average was: 1329.

It gave a profit of $6531620.13.

Which gave a percent increase of 6531.62013%.

The Constant Investment method gave a profit of $549648.6000000054.

Which gave a percent increase of 549.6486000000054%.

The 'buy and hold' method gave a profit of $1.1189285714285713E7.

Which was a percent increase of 11189.285714285712%.

Using these strategies, the following results were achieved between 1999-04-22 and 2009-03-30.

The best moving average was: 1266.

It now gives a profit of $21579.67999999995.

Which gives a percent increase of 21.57967999999995%.

The best weighted average was: 1329.

It now gives a profit of $-3015.3299999999726.

Which gives a percent increase of -3.0153299999999725%.

The Constant Investment method now gives a profit of $117506.28999999951.

Which gives a percent increase of 117.50628999999951%.

The 'buy and hold' method now gives a profit of $115749.76422508643.

Which gives a percent increase of 115.74976422508642%.

Here is the output so far from the Fourier method.

```
Best-fit line: 1.0169389475913724 + 5.153199303153387E-4 X^1

The residuals:

-0.014515468068378778
-0.009094169369208815
-0.008578849438893421
-0.008063529508578249
-0.012454188347117423
-0.0037488144467576756
0.018379981904094844
0.005539510652854629
0.009377092775147755
-0.0016875347254963202
0.003774209140144702
-0.008846948834910107
-0.011588960774901347
-0.011073640844585952
0.009146750165061723
0.02638486782483751
```

This then continues outputting residuals for every data point we have for that stock. It then repeats and gives the best-fit line and the residuals for every other stock.

Analysis

The stocks used were chosen at random from a list of many stocks in different sectors. The well-being of each stock should be independent, and, therefore, the stock prices for separate stocks are as independent as possible. For each pair of methods, the histograms for the differences between percent increases were symmetric and unimodal, and therefore, nearly normal. The percent increases given by the various methods for each stock are dependent because the well-being of each company affects the profit regardless of the method used. The number of stocks in the sample, 20, is also much less than 10% of the total number of stocks. All of the assumptions hold true for a paired t-test. So, we used a t-test on the difference between the percent increases for different methods.

Alpha = .05. So to reject our null hypotheses, the p-values must be less than .05.

Moving average vs. Weighted moving average:

Null hypothesis: the moving average method gives the same mean percent increase as the weighted moving average method

Alternative hypothesis: the weighted gives a better mean percent increase than the simple moving average

Results:

p = .378523

So, assuming that the moving average gives the same percent increase as the weighted moving average, there is a 37.8523% chance of obtaining our data. Therefore, because p

> .05, we cannot reject our null hypothesis. There is no evidence that the weighted moving average gives a greater percent increase than the moving average.

Moving average vs. Constant investment:

Null hypothesis: the moving average method gives the same mean percent increase as the constant investment method

Alternative hypothesis: the constant investment method gives a better mean percent increase than the moving average method.

Results:

P = 1.2413 X 10^-4

Assuming that the moving average method gives the same percent increase as the constant investment method, there is a .012412% chance of obtaining our data. Because p < .05, we reject the null hypothesis. Our data provides evidence that the constant investment method gives a greater percent increase than the moving average method.

Moving average vs. Buy and hold:

Null Hypothesis: the moving average method gives the same mean percent increase as the buy and hold method

Alternative Hypothesis: the buy and hold method gives a better mean percent increase than the moving average method

Results:

P = .0399.

Assuming that the moving average method gives the same percent increase as the buy and hold method, there is a 3.99% chance of obtaining our data. Because p < .05, we reject the null hypothesis. There is evidence that the buy and hold method gives a greater percent increase than the moving average method.

Buy and hold vs. Constant investment:

Null hypothesis: the buy and hold method gives the same mean percent increase as the constant investment method

Alternative hypothesis: the constant investment method gives a greater mean percent increase than the buy and hold method

Results:

P = 1.13 X 10^-5

Assuming that the buy and hold method gives the same percent increase as the constant investment method, there is a .00113% chance of obtaining our data. Because p < .05, we reject the null hypothesis. There is evidence that the constant investment method gives a greater percent increase than the buy and hold method.

From these statistical tests, we see that the constant investment method gives the greatest percent increase on average, followed by the buy and hold method. The moving average and weighted moving average methods are the worst; however, there is no evidence that one method is better than the other.

Conclusion

We showed that it is hard to predict whether or not a moving average or a weighted average will work well. It also shows that although a moving or weighted average worked well during one period, it is not necessarily true over the lifetime of the stock. These commonly methods do not give any noticeable advantage over just a buy and hold method. We also found that the constant investment method worked best. Buying lower and selling higher works well, and most stocks don’t completely crash, so you don’t have to worry about catching a falling knife. Stocks that drop love to return, and this allows you to always have a chance in getting a part of that rebound.

In addition, we observed patterns while looking at our data. For example, whenever the percent increase is very high, the constant investment method only gave moderate returns. However, lower returns for methods like buy and hold would often still give a profit for constant investment. This shows that constant invest is also less risky than many of the other methods.

However, there are some problems with comparing the results straight out. With the constant investment, it is possible that you will not have enough money to keep buying stock. Also, with all of the methods except for buy and hold. There are often times when not all of ones money is being invested. There should be some compensation for this, because the money would, in reality, be invested in another stock. However, it can be reasoned that these compensations would not affect our results much, or would just reinforce our results more.

Recommendations

We want to change the program to give us better, more reliable results. Including how much money was invested and for how long can give us a good base for comparing methods completely equally. We also want to include methods to help us determine the risk and reward of each method overall. A program to investigate other ways of choosing moving averages (like the best between 20 days and 200 days instead of having lots of 2s, 3s, 4s, and 1000+).

Finally, we want to try to finish our Fourier work. We haven’t found a good way to extrapolate, because most methods we found are periodic over the entire region, which would just cause the value we had for the first day to show up for the next day. However, there is still doubt as to whether this will predict anything in a timely and day-to-day fashion.

There are also other minor changes, like including more stocks and being able to input a stock for testing (user input). Graphics are also being looked at, but the main program is the real heart of the project.

Acknowledgements

We would like to thank our computer science instructor, sponsoring teacher, and mentor, Mr. Mims, for encouraging us to participate in the New Mexico Supercomputing Challenge and for continually giving us feedback on our presentations and our code. Most of all, we would like to thank everyone who makes the supercomputing challenge possible. We have thoroughly enjoyed our participation in the challenge, and we believe it has taught us many skills that will help us in college and in our future careers.

References

Internet Sources:

BigCharts: Stock Charts, Screeners, Interactive Charting and Research Tools. 19 Dec. 2008 http://www.bigcharts.com.

Google Finance. 19 Dec. 2008 http://finance.google.com/finance.

"Moving Averages - StockCharts.com." StockCharts.com - Simply the Web's Best Stock Charts. 19 Dec. 2008 http://stockcharts.com/school/doku.php?id=chart_school:technical_indicators:moving_averages.

Yahoo! Finance. 19 Dec. 2008 http://finance.yahoo.com/.

Books:

Besset, Didier H. Object-oriented implementation of numerical methods an introduction with Java and Smalltalk. San Francisco: Morgan Kaufmann, 2001.

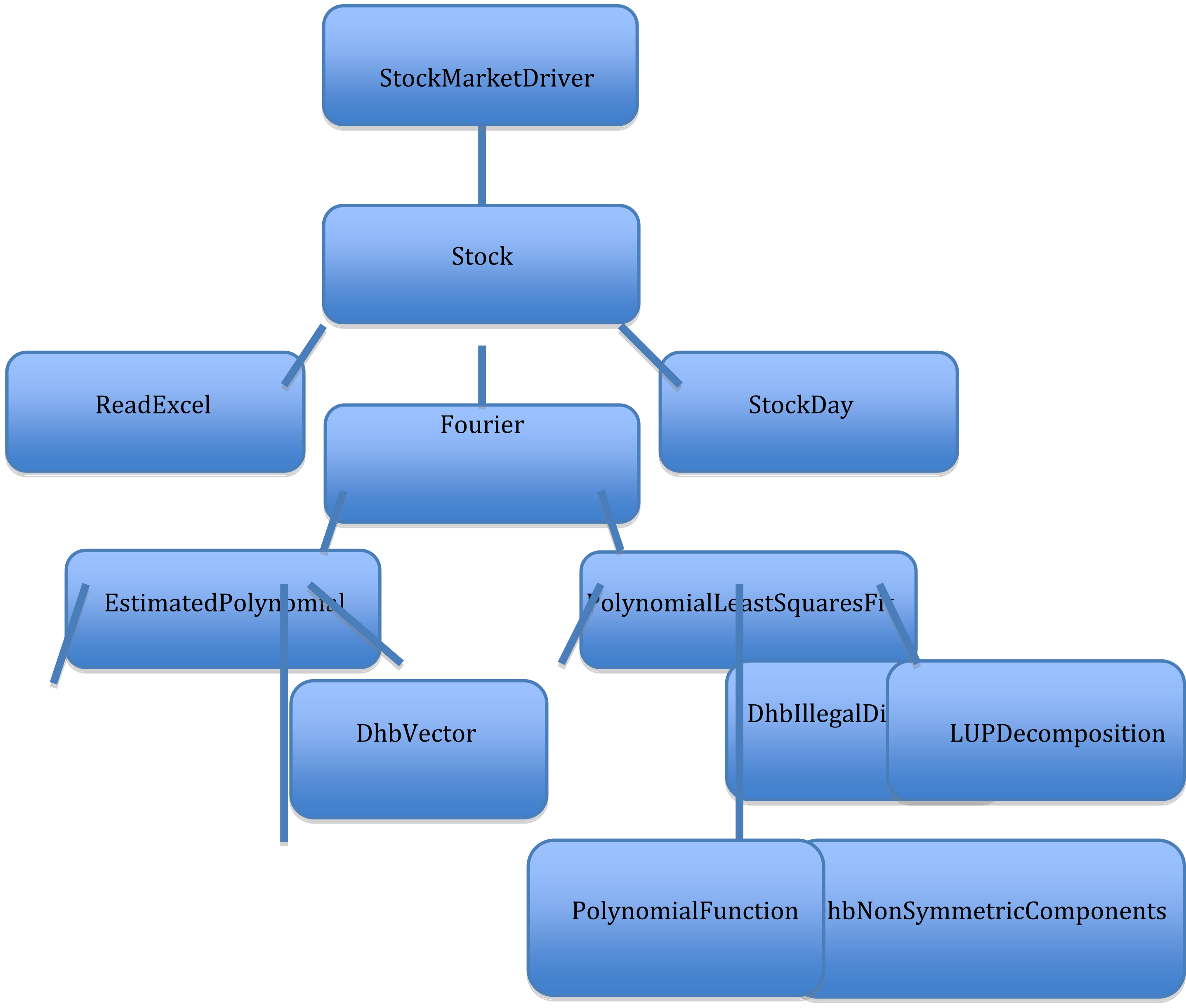
Edwards, C. H., and David E. Penney. Elementary Differential Equations With Boundary Value Problems. 6th ed. Upper Saddle River, New Jersey: Pearson Education, Inc., 2008.

Joshi, Mark S., Mark Broadie, and Sam Howison. The Concepts and Practice of Mathematical Finance. New York: Cambridge UP, 2003.

Morris, Kenneth M., and Virginia B. Morris. The Wall Street Journal Guide to Understanding Money and Investing. New York: Fireside, 1994.

Roehner, Bertrand M. Patterns of Speculation : A Study in Observational Econophysics. New York: Cambridge UP, 2005.

Appendix A: Code Flow Diagram



Appendix B: StockMarketDriver

```
import java.util.*;

public class StockMarketDriver
{
	public static void main(String[] args)
	{
		ArrayList <Stock> Stocks = new ArrayList <Stock> ();
		Stocks.add(new Stock ("BAC"));
		Stocks.add(new Stock ("CMCSA"));
		Stocks.add(new Stock ("COST"));
		Stocks.add(new Stock ("DE"));
		Stocks.add(new Stock ("DOW"));
		Stocks.add(new Stock ("EXC"));
		Stocks.add(new Stock ("F"));
		Stocks.add(new Stock ("GE"));
		Stocks.add(new Stock ("GIS"));
		Stocks.add(new Stock ("HNZ"));
		Stocks.add(new Stock ("JAVA"));
		Stocks.add(new Stock ("JNJ"));
		Stocks.add(new Stock ("LMT"));
		Stocks.add(new Stock ("LUV"));
		Stocks.add(new Stock ("MCD"));
		Stocks.add(new Stock ("MOT"));
		Stocks.add(new Stock ("SLM"));
		Stocks.add(new Stock ("SUN"));
		Stocks.add(new Stock ("WMT"));
		Stocks.add(new Stock ("XOM"));
		for (int i = 0; i < Stocks.size(); i++)
		{
			Stocks.get(i).retrieveData();
			Stocks.get(i).outputData();
			Stocks.get(i).fourier();
		}
	}
}
```

Appendix C: StockDay

```
import jxl.Cell;

public class StockDay
{
	private String date;
	private double open;
	private double high;
	private double low;
	private double close;
	private double ln;
	private double volume;
	private double adjClose;

	public StockDay()
	{

	}

	public void Add(Cell cell, int i)
	{
		String contents = cell.getContents();
		switch (i)
		{
			case 0:
				date = contents;
				break;
			case 1:
				open = Double.valueOf(contents).doubleValue();
				break;
			case 2:
				high = Double.valueOf(contents).doubleValue();
				break;
			case 3:
				low = Double.valueOf(contents).doubleValue();
				break;
			case 4:
				close = Double.valueOf(contents).doubleValue();
				break;
			case 5:
				volume = Integer.valueOf(contents);
				break;
			case 6:
				adjClose = Double.valueOf(contents).doubleValue();
				ln = Math.log(adjClose);
		}
	}

	public void output()
	{
		System.out.println(date + " " + open + " " + high + " " + low + " " + close + " " + 
volume);
	}
```

```
        public String getDate()
        {
                return date;
        }

        public double getClose()
        {
                return close;
        }

        public double getAdjClose()
        {
                return adjClose;
        }

        public double getLn()
        {
                return ln;
        }
}
```

Appendix D: ReadExcel

```
import java.io.*;
import jxl.Cell;
import jxl.Sheet;
import jxl.Workbook;
import jxl.read.biff.BiffException;
import java.util.*;

public class ReadExcel
{
	public static void read()
	{
		read("GOOG");
	}
	public static ArrayList <StockDay> read(String stockTag)
	{
		ArrayList<StockDay> data = new ArrayList <StockDay> ();

		try
		{
			StockDay entry;
			File file = new File("References/" + stockTag + ".xls");
			Workbook wb = Workbook.getWorkbook(file);
			Sheet sheet = wb.getSheet(0);

			try
			{
				for(int i=1; i>0; i++)		// Not good programming, but it works.
				{
					entry = new StockDay();
					for(int j=0; j<=6; j++)
					{
						Cell cell = sheet.getCell(j,i);
						entry.Add(cell, j);
					}
					data.add(0, entry);
				}
			}
			catch(Exception e){}
			wb.close();

		}

		catch(IOException e){}
		catch(BiffException e){}
		finally{}

		return data;
	}

	public static void print_cells(Cell printing, int k, int l)
	{
		if(l==0)
		{
			String newcell = printing.getContents();
```

```
                        System.out.print(newcell+" ");
                }

                else
                {
                        String newcell = printing.getContents();
                        double numbercell = Double.valueOf(newcell).doubleValue();
                        System.out.print(numbercell+" ");
                }

                if(l%5==0 && l!=0)
                {
                        System.out.println();
                }
        }
}
```

Appendix E: Stock

```
import java.util.*;

public class Stock
{
	private ArrayList <StockDay> entries;
	private String tag;
	private Fourier freq;

	public Stock()
	{
		entries = new ArrayList <StockDay> ();
		//freq = new Fourier (entries);
	}
	public Stock(String t)
	{
		entries = new ArrayList <StockDay> ();
		//freq = new Fourier (entries);
		tag = t;
	}

	public void retrieveData()
	{
		entries = ReadExcel.read(tag);
	}

	public void outputData()
	{
		System.out.println();
		System.out.println("Data for " + tag + ".");
		/*System.out.println("Date Open High Low Close Volume AdjustedClose");
		for (int i = 0; i < entries.size(); i++)
		{
			entries.get(i).output();
		}*/	//	Extra output
		System.out.println("Size: " + entries.size());
		System.out.println();

		//Find
		System.out.print("For " + entries.get(0).getDate() + " until ");
		System.out.println(entries.get(entries.size() - 2501).getDate() + ", these were the best
results.");

		//	Moving Average
		int bestMA = findBestMovingAverage(0, entries.size() - 2500);
		double profitMA1 = useMovingAverage(0, entries.size() - 2500, bestMA);
		System.out.println("The best moving average was: " + bestMA + ".");
		System.out.println("It gave a profit of $" + profitMA1 + ".");
		System.out.println("Which gave a percent increase of " + profitMA1 / 1000 + "%.");

		//	Weighted Average
		int bestWA = findBestWeightedAverage(0, entries.size() - 2500);
		double profitWA1 = useWeightedAverage(0, entries.size() - 2500, bestWA);
		System.out.println("The best weighted average was: " + bestWA + ".");
		System.out.println("It gave a profit of $" + profitWA1 + ".");
		System.out.println("Which gave a percent increase of " + profitWA1 / 1000 + "%.");

		//		Constant Investment
		double profitCI1 = useConstantInvestment(0, entries.size() - 2500);
		System.out.println("The Constant Investment method gave a profit of $" + profitCI1 +
".");

		System.out.println("Which gave a percent increase of " + profitCI1 / 1000 + "%.");

		//		Buy and Hold
		double buyHoldProfit1 = useBuyAndHold(0, entries.size() - 2500);
		System.out.println("The 'buy and hold' method gave a profit of $" + buyHoldProfit1 +
".");

		System.out.println("Which was a percent increase of " + buyHoldProfit1 / 1000 + "%.");
		System.out.println();

		//		Test
		System.out.print("Using these strategies, the following results were achieved between ");
		System.out.println(entries.get(entries.size() - 2500).getDate() + " and " +
entries.get(entries.size() - 1).getDate() + ".");

		//		Moving Average
		double profitMA2 = useMovingAverage(entries.size() - 2500, entries.size(), bestMA);
		System.out.println("The best moving average was: " + bestMA + ".");
		System.out.println("It now gives a profit of $" + profitMA2 + ".");
		System.out.println("Which gives a percent increase of " + profitMA2 / 1000 + "%.");

		//		Weighted Average
		double profitWA2 = useWeightedAverage(entries.size() - 2500, entries.size(), bestWA);
		System.out.println("The best weighted average was: " + bestWA + ".");
		System.out.println("It now gives a profit of $" + profitWA2 + ".");
		System.out.println("Which gives a percent increase of " + profitWA2 / 1000 + "%.");

		//		Constant Investment
		double profitCI2 = useConstantInvestment(entries.size() - 2500, entries.size());
		System.out.println("The Constant Investment method now gives a profit of $" + profitCI2
+ ".");

		System.out.println("Which gives a percent increase of " + profitCI2 / 1000 + "%.");

		//		Buy and Hold
		double buyHoldProfit2 = useBuyAndHold(entries.size() - 2500, entries.size());
		System.out.println("The 'buy and hold' method now gives a profit of $" + buyHoldProfit2
+ ".");

		System.out.println("Which gives a percent increase of " + buyHoldProfit2 / 1000 +
"%.");

		System.out.println();
	}

	public double movingAverage(int day, int timeFrame)
	{
		double sum = 0;
		for (int i = Math.max(0, day - timeFrame + 1); i <= day; i++)
		{
			sum += entries.get(i).getAdjClose();
		}
		return sum/timeFrame;
	}

	public double movingAverage(int day, int timeFrame, double prevAverage)
	{
		if (prevAverage == 0)
		{
			return movingAverage(day, timeFrame);
		}
		else
		{
			double average = prevAverage + (entries.get(day).getAdjClose() - 
entries.get(day - timeFrame).getAdjClose()) / timeFrame;
			return average;
		}
	}

	public int findBestMovingAverage()
	{
		return findBestMovingAverage(0, entries.size());
	}

	public int findBestMovingAverage(int start, int end)
	{
		int bestTimeFrame = 1;
		double bestProfit = 0;
		double profit = 0;
		for (int i = 2; i < end - start; i++)
		{
			profit = useMovingAverage(start, end, i);
			if (profit > bestProfit)
			{
				bestProfit = profit;
				bestTimeFrame = i;
				//System.out.println(i + " $" + profit);		// Debug
			}
			/*if (i % 100 == 0)
			{
				System.out.println(i);
			}*/	//	Debug
		}
		return bestTimeFrame;
	}

	public double useMovingAverage(int start, int end, int timeFrame)
	{
		double money = 100000;
		int shares = 0;
		double tempAverage = 0;
		double tempPrice = 0;
		for (int i = timeFrame + start - 1; i < end; i++)
		{
			tempPrice = entries.get(i).getAdjClose();
			money += shares * tempPrice;
			shares = 0;
			//tempAverage = movingAverage(i, timeFrame);	//		Old Version
08/01/08
			tempAverage = movingAverage(i, timeFrame, tempAverage);	//		New
Version
			if (tempAverage < tempPrice)
			{
				shares = (int)(money / tempPrice);
				money -= shares * tempPrice;
			}
		}
		money += shares * tempPrice;
		return money - 100000;
	}

	public double weightedAverage(int day, int timeFrame)
	{
		double sum = 0;
		for (int i = Math.max(0, day - timeFrame + 1); i <= day; i++)
		{
			sum += entries.get(i).getAdjClose()*(i + timeFrame - day);
		}
		return 2*sum/timeFrame/(timeFrame+1);
	}

	public double weightedAverage(int day, int timeFrame, double prevMoving, double
prevWeighted)
	{
		if (prevWeighted == 0)
		{
			return weightedAverage(day, timeFrame);
		}

		else
		{
			double average = prevWeighted + (entries.get(day).getAdjClose() -
prevMoving) * 2 / (timeFrame + 1);
			return average;
		}
	}

	public int findBestWeightedAverage()
	{
		return findBestWeightedAverage(0, entries.size());
	}

	public int findBestWeightedAverage(int start, int end)
	{
		int bestTimeFrame = 1;
		double bestProfit = 0;
		double profit = 0;
		for (int i = 2; i < end - start; i++)
		{
			profit = useWeightedAverage(start, end, i);
			if (profit > bestProfit)
			{
```

```
                        bestProfit = profit;
                        bestTimeFrame = i;
                        //System.out.println(i + " $" + profit);        //debug
                    }
                    /*if (i % 100 == 0)
                    {
                        System.out.println(i);
                    }*/ //debug
                }
                return bestTimeFrame;
            }

            public double useWeightedAverage(int start, int end, int timeFrame)
            {
                double money = 100000;
                int shares = 0;
                double tempWeight = 0;
                double tempMoving = 0;
                double tempPrice = 0;
                for (int i = timeFrame + start - 1; i < end; i++)
                {
                    tempPrice = entries.get(i).getAdjClose();
                    money += shares * tempPrice;
                    shares = 0;
                    tempWeight = weightedAverage(i, timeFrame, tempMoving, tempWeight);
                    if (tempWeight < tempPrice)
                    {
                        shares = (int)(money / tempPrice);
                        money -= shares * tempPrice;
                    }
                    tempMoving = movingAverage(i, timeFrame, tempMoving);
                }
                money += shares * tempPrice;
                return money - 100000;
            }

            public double useConstantInvestment()
            {
                return useConstantInvestment(0, entries.size());
            }

            public double useConstantInvestment(int start, int end)       //      Start is inclusive, end is
exclusive
            {
                double money = 100000;
                int shares = 0;
                double tempPrice = 0;
                for (int i = start; i < end; i++)
                {
                    tempPrice = entries.get(i).getAdjClose();
                    money += shares * tempPrice;
                    shares = 0;
                    shares = (int)(100000 / tempPrice);
                    money -= shares * tempPrice;
                }
                money += shares * tempPrice;
```

```
			return money - 100000;
		}

		public double useBuyAndHold()
		{
			return useBuyAndHold(0, entries.size());
		}

		public double useBuyAndHold(int start, int end)	//		Start is inclusive, end is exclusive
		{
			return entries.get(end-1).getAdjClose()/entries.get(start).getAdjClose()*100000 - 
	100000;
		}

		public void fourier()
		{
			freq = new Fourier (entries);
			freq.calculate();
		}
										}
```

Appendix F: Fourier

```
import java.util.ArrayList;

public class Fourier
{
	private ArrayList <StockDay> entries;
	double residual[];

	public Fourier(ArrayList <StockDay> e)
	{
		entries = e;
		residual = new double[e.size()];
	}

	public void calculate()
	{
		//first order polynomial
		PolynomialLeastSquareFit fit = new PolynomialLeastSquareFit(1);

		//least-squares polynomial fit
		for(int i = 0; i< entries.size(); i++)
		{
			fit.accumulatePoint(i, entries.get(i).getLn());
		}
		EstimatedPolynomial estimation = fit.evaluate();
		System.out.println("Best-fit line: " + estimation);


		//finds residuals
		System.out.println("The residuals:");
		for(int i = 0; i<entries.size(); i++)
		{
			residual[i] = estimation.value(i) - entries.get(i).getLn();
			System.out.println(residual[i]);
		}
	}

	/*public void FastFourier()
	{
		This method will find an and bn to be used to make a Fourier series
		that approximates the residual plot.
	}*/
}
```

Appendix G: PolynomialFunction

```
import java.util.Vector;
import java.util.Enumeration;
public class PolynomialFunction implements OneVariableFunction
{
            /**
             * Polynomial coefficients.
             */
            private double[] coefficients;


/**
 * Constructor method.
 * @param coeffs polynomial coefficients.
 */
public PolynomialFunction( double[] coeffs)
{
            coefficients = coeffs;
}
/**
 *
 * @param r double         number added to the polynomial.
 * @return DhbFunctionEvaluation.PolynomialFunction
 */
public PolynomialFunction add( double r)
{
            int n = coefficients.length;
            double coef[] = new double[n];
            coef[0] = coefficients[0] + r;
            for ( int i = 1; i < n; i++)
                        coef[i] = coefficients[i];
            return new PolynomialFunction( coef);
}
/**
 *
 * @param p DhbFunctionEvaluation.PolynomialFunction
 * @return DhbFunctionEvaluation.PolynomialFunction
 */
public PolynomialFunction add( PolynomialFunction p)
{
            int n = Math.max( p.degree(), degree()) + 1;
            double[] coef = new double[n];
            for ( int i = 0; i < n; i++ )
                        coef[i] = coefficient(i) + p.coefficient(i);
            return new PolynomialFunction( coef);
}
/**
 * Returns the coefficient value at the desired position
 * @param n int  the position of the coefficient to be returned
 * @return double the coefficient value
 * @version 1.2
 */
```

```
public double coefficient( int n)
{
            return n < coefficients.length ? coefficients[n] : 0;
}
/**
 *
 * @param r double          a root of the polynomial (no check made).
 * @return PolynomialFunction the receiver divided by polynomial (x - r).
 */
public PolynomialFunction deflate( double r)
{
            int n = degree();
            double remainder = coefficients[n];
            double[] coef = new double[n];
            for ( int k = n - 1; k >= 0; k--)
            {
                        coef[k] = remainder;
                        remainder = remainder * r + coefficients[k];
            }
            return new PolynomialFunction( coef);
}
/**
 * Returns degree of this polynomial function
 * @return int degree of this polynomial function
 */
public int degree()
{
            return coefficients.length - 1;
}
/**
 * Returns the derivative of the receiver.
 * @return PolynomialFunction derivative of the receiver.
 */
public PolynomialFunction derivative()
{
            int n = degree();
            if ( n == 0 )
            {
                        double coef[] = {0};
                        return new PolynomialFunction( coef);
            }
            double coef[] = new double[n];
            for ( int i = 1; i <= n; i++)
                        coef[i-1] = coefficients[i]*i;
            return new PolynomialFunction( coef);
}
/**
 *
 * @param r double
 * @return DhbFunctionEvaluation.PolynomialFunction
 */
public PolynomialFunction divide( double r)
{
            return multiply( 1 / r);
}
/**
```

```
 *
 * @param r double
 * @return DhbFunctionEvaluation.PolynomialFunction
 */
public PolynomialFunction divide( PolynomialFunction p)
{
		return divideWithRemainder(p)[0];
}
/**
 *
 * @param r double
 * @return DhbFunctionEvaluation.PolynomialFunction
 */
public PolynomialFunction[] divideWithRemainder( PolynomialFunction p)
{
		PolynomialFunction[] answer = new PolynomialFunction[2];
		int m = degree();
		int n = p.degree();
		if ( m < n )
		{
				double[] q = {0};
				answer[0] = new PolynomialFunction( q);
				answer[1] = p;
				return answer;
		}
		double[] quotient = new double[ m - n + 1];
		double[] coef = new double[ m + 1];
		for ( int k = 0; k <= m; k++ )
				coef[k] = coefficients[k];
		double norm = 1 / p.coefficient( n);
		for ( int k = m - n; k >= 0; k--)
		{
				quotient[k] = coef[ n + k] * norm;
				for ( int j = n + k - 1; j >= k; j--)
						coef[j] -= quotient[k] * p.coefficient(j-k);
		}
		double[] remainder = new double[n];
		for ( int k = 0; k < n; k++)
				remainder[k] = coef[k];
		answer[0] = new PolynomialFunction( quotient);
		answer[1] = new PolynomialFunction( remainder);
		return answer;
}
/**
 * Returns the integral of the receiver having the value 0 for X = 0.
 * @return PolynomialFunction integral of the receiver.
 */
public PolynomialFunction integral( )
{
		return integral( 0);
}
/**
 * Returns the integral of the receiver having the specified value for X = 0.
 * @param value double   value of the integral at x=0
 * @return PolynomialFunction integral of the receiver.
 */
```

```
public PolynomialFunction integral( double value)
{
		int n = coefficients.length + 1;
		double coef[] = new double[n];
		coef[0] = value;
		for ( int i = 1; i < n; i++)
			coef[i] = coefficients[i-1]/i;
		return new PolynomialFunction( coef);
}
/**
 *
 * @param r double
 * @return DhbFunctionEvaluation.PolynomialFunction
 */
public PolynomialFunction multiply( double r)
{
		int n = coefficients.length;
		double coef[] = new double[n];
		for ( int i = 0; i < n; i++)
			coef[i] = coefficients[i] * r;
		return new PolynomialFunction( coef);
}
/**
 *
 * @param p DhbFunctionEvaluation.PolynomialFunction
 * @return DhbFunctionEvaluation.PolynomialFunction
 */
public PolynomialFunction multiply( PolynomialFunction p)
{
		int n = p.degree() + degree();
		double[] coef = new double[n + 1];
		for ( int i = 0; i <= n; i++)
		{
			coef[i] = 0;
			for ( int k = 0; k <= i; k++)
				coef[i] += p.coefficient(k) * coefficient(i-k);
		}
		return new PolynomialFunction( coef);
}
/**
 *
 * @return double[]
 */
public double[] roots()
{
		return roots( DhbMath.defaultNumericalPrecision());
}
/**
 *
 * @param desiredPrecision double
 * @return double[]
 */
public double[] roots( double desiredPrecision)
{
		PolynomialFunction dp = derivative();
		double start = 0;
		while ( Math.abs( dp.value( start)) < desiredPrecision )
			start = Math.random();
		PolynomialFunction p = this;
		NewtonZeroFinder rootFinder = new NewtonZeroFinder( this, dp, start);
		rootFinder.setDesiredPrecision( desiredPrecision);
		Vector rootCollection = new Vector( degree());
		while ( true)
		{
			rootFinder.evaluate();
			if ( !rootFinder.hasConverged() )
				break;
			double r = rootFinder.getResult();
			rootCollection.addElement( new Double( r));
			p = p.deflate( r);
			if ( p.degree() == 0 )
				break;
			rootFinder.setFunction( p);
			try { rootFinder.setDerivative( p.derivative());}
					catch ( IllegalArgumentException e) {};
		}
		double[] roots = new double[ rootCollection.size()];
		Enumeration e = rootCollection.elements();
		int n = 0;
		while ( e.hasMoreElements() )
		{
			roots[n++] = ( (Double) e.nextElement()).doubleValue();
		}
		return roots;
	}
	/**
	 *
	 * @param p DhbFunctionEvaluation.PolynomialFunction
	 * @return DhbFunctionEvaluation.PolynomialFunction
	 */
	public PolynomialFunction subtract( double r)
	{
		return add( -r);
	}
	/**
	 *
	 * @return DhbFunctionEvaluation.PolynomialFunction
	 * @param p DhbFunctionEvaluation.PolynomialFunction
	 */
	public PolynomialFunction subtract( PolynomialFunction p)
	{
		int n = Math.max( p.degree(), degree()) + 1;
		double[] coef = new double[n];
		for ( int i = 0; i < n; i++ )
			coef[i] = coefficient(i) - p.coefficient(i);
		return new PolynomialFunction( coef);
	}
	/**
	* Returns a string representing the receiver
	*/
	public String toString()
	{
        StringBuffer sb = new StringBuffer();
        boolean firstNonZeroCoefficientPrinted = false;
        for ( int n = 0; n < coefficients.length; n++)
        {
                if ( coefficients[n] != 0 )
                {
                        if ( firstNonZeroCoefficientPrinted)
                                sb.append( coefficients[n] > 0 ? " + " : " ");
                        else
                                firstNonZeroCoefficientPrinted = true;
                        if ( n == 0 || coefficients[n] != 1)
                                sb.append( Double.toString( coefficients[n]) );
                        if ( n > 0 )
                                sb.append( " X^"+n);
                }
        }
        return sb.toString();
}
/**
 * Returns the value of the polynomial for the specified variable value.
 * @param x double        value at which the polynomial is evaluated
 * @return double polynomial value.
 */
public double value( double x)
{
        int n = coefficients.length;
        double answer = coefficients[--n];
        while ( n > 0 )
                answer = answer * x + coefficients[--n];
        return answer;
}
/**
 * Returns the value and the derivative of the polynomial
 * for the specified variable value in an array of two elements
 * @version 1.2
 * @param x double        value at which the polynomial is evaluated
 * @return double[0]   the value of the polynomial
 * @return double[1]   the derivative of the polynomial
 */
public double[] valueAndDerivative( double x)
{
        int n = coefficients.length;
        double[] answer = new double[2];
        answer[0] = coefficients[--n];
        answer[1] = 0;
        while ( n > 0 )
        {
                answer[1] = answer[1] * x + answer[0];
                answer[0] = answer[0] * x + coefficients[--n];
        }
        return answer;
}
}
```

Appendix H: PolynomialLeastSquareFit

```
public class PolynomialLeastSquareFit
{
		double[][] systemMatrix;
		double[] systemConstants;
/**
 * Constructor method.
 */
public PolynomialLeastSquareFit( int n)
{
		int n1 = n + 1;
		systemMatrix = new double[n1][n1];
		systemConstants = new double[n1];
		reset();
}
/**
 * @param x double
 * @param m StatisticalMoments
 */
public void accumulateAverage( double x, StatisticalMoments m)
{
		accumulatePoint( x, m.average(), m.errorOnAverage());
}
/**
 * @param x double
 * @param n int  bin content
 */
public void accumulateBin( double x, int n)
{
		accumulateWeightedPoint( x, n, 1.0 / Math.max( 1, n));
}
/**
 * @param x double
 * @param y double
 */
public void accumulatePoint( double x, double y)
{
		accumulateWeightedPoint( x, y, 1);
}
/**
 * @param x double
 * @param y double
 * @param error double    standard deviation on y
 */
public void accumulatePoint( double x, double y, double error)
{
		accumulateWeightedPoint( x, y, 1.0 / (error * error));
}
/**
 * @param x double
 * @param y double
 * @param w double        weight of point
 */
public void accumulateWeightedPoint( double x, double y, double w)
{
	double xp1 = w;
	double xp2;
	for ( int i = 0; i < systemConstants.length; i++ )
	{
		systemConstants[i] += xp1 * y;
		xp2 = xp1;
		for ( int j = 0; j <= i; j++ )
		{
			systemMatrix[i][j] += xp2;
			xp2 *= x;
		}
		xp1 *= x;
	}
}
/**
 * @return DhbEstimation.EstimatedPolynomial
 */
public EstimatedPolynomial evaluate()
{
	for ( int i = 0; i < systemConstants.length; i++ )
	{
		for ( int j = i + 1; j < systemConstants.length; j++ )
		{
			systemMatrix[i][j] = systemMatrix[j][i];
		}
	}

	try {
		LUPDecomposition lupSystem = new LUPDecomposition(

		systemMatrix);
		double [][] components = lupSystem.inverseMatrixComponents();
		LUPDecomposition.symmetrizeComponents( components);
		return new EstimatedPolynomial(
						lupSystem.solve( systemConstants),
						SymmetricMatrix.fromComponents( components));
		} catch ( DhbIllegalDimension e) {System.out.println("Illegal Dimension");}
		  catch ( DhbNonSymmetricComponents ex) {System.out.println("NonSymmetric
Components");};
	 return null;
}
public void reset()
{
	for ( int i = 0; i < systemConstants.length; i++ )
	{
		systemConstants[i] = 0;
		for ( int j = 0; j < systemConstants.length; j++ )
			systemMatrix[i][j] = 0;
	}
}
						}
```

Appendix I: EstimatedPolynomial

```
public class EstimatedPolynomial extends PolynomialFunction
{
        /**
         * Error matrix.
         */
        SymmetricMatrix errorMatrix;
/**
 * Constructor method.
 * @param coeffs double[]
 * @param e double[]      error matrix
 */
public EstimatedPolynomial(double[] coeffs, SymmetricMatrix e)
{
        super(coeffs);
        errorMatrix = e;
}
/**
 * @return double
 * @param x double
 */
public double error( double x)
{
        int n = degree() + 1;
        double[] errors = new double[n];
        errors[0] = 1;
        for ( int i = 1; i < n; i++)
                errors[i] = errors[i-1] * x;
        DhbVector errorVector = new DhbVector( errors);
        double answer;
        try { answer = errorVector.product(
                                                        errorMatrix.product( errorVector));
                } catch (DhbIllegalDimension e) { answer = Double.NaN;};
        return Math.sqrt( answer);
}

                                        }
```

Appendix J: DhbVector

```
public class DhbVector
{
            protected double[] components;

/**
 * Create a vector of given dimension.
 * NOTE: The supplied array of components must not be changed.
 * @param comp double[]
 */
public DhbVector(  double comp[]) throws NegativeArraySizeException
{
            int n = comp.length;
            if ( n <= 0 )
                        throw new NegativeArraySizeException(
                                                                                    "Vector components cannot be
empty");
            components = new double[n];
            System.arraycopy( comp, 0, components, 0, n);
}
/**
 * Create a vector of given dimension.
 * @param dimension int dimension of the vector; must be positive.
 */
public DhbVector ( int dimension) throws NegativeArraySizeException
{
            if ( dimension <= 0 )
                        throw new NegativeArraySizeException(
                                                                                    "Requested vector size:
"+dimension);
            components = new double[dimension];
            clear();
}
/**
 * @param v DHBmatrixAlgebra.DhbVector
 * @exception DHBmatrixAlgebra.DhbIllegalDimension if the vector
 * and supplied vector do not have the same dimension.
 */
public void accumulate ( double[] x) throws DhbIllegalDimension
{
            if ( this.dimension() != x.length )
                        throw new DhbIllegalDimension("Attempt to add a "
                                                      +this.dimension()+"-dimension vector to a "
                                                                                    +x.length+"-dimension
array");
            for ( int i = 0; i < this.dimension(); i++)
                        components[i] += x[i];
}
/**
 * @param v DHBmatrixAlgebra.DhbVector
 * @exception DHBmatrixAlgebra.DhbIllegalDimension if the vector
 * and supplied vector do not have the same dimension.
 */
public void accumulate ( DhbVector v) throws DhbIllegalDimension
{
		if ( this.dimension() != v.dimension() )
			throw new DhbIllegalDimension("Attempt to add a "
						+this.dimension()+"-dimension vector to a "
								+v.dimension()+"-dimension
vector");
		for ( int i = 0; i < this.dimension(); i++)
			components[i] += v.components[i];
}
/**
 * @param v DHBmatrixAlgebra.DhbVector
 * @exception DHBmatrixAlgebra.DhbIllegalDimension if the vector
 * and supplied vector do not have the same dimension.
 */
public void accumulateNegated( double[] x) throws DhbIllegalDimension
{
		if ( this.dimension() != x.length )
			throw new DhbIllegalDimension("Attempt to add a "
							+this.dimension()+"-dimension vector to a "
										+x.length+"-
dimension array");
		for ( int i = 0; i < this.dimension(); i++)
			components[i] -= x[i];
}
/**
 * @param v DHBmatrixAlgebra.DhbVector
 * @exception DHBmatrixAlgebra.DhbIllegalDimension if the vector
 * and supplied vector do not have the same dimension.
 */
public void accumulateNegated( DhbVector v) throws DhbIllegalDimension
{
		if ( this.dimension() != v.dimension() )
			throw new DhbIllegalDimension("Attempt to add a "
							+this.dimension()+"-dimension vector to a "
								+v.dimension()+"-dimension
vector");
		for ( int i = 0; i < this.dimension(); i++)
			components[i] -= v.components[i];
}
/**
 * @return DHBmatrixAlgebra.DhbVector sum of the vector with
 *
		the supplied vector
 * @param v DHBmatrixAlgebra.DhbVector
 * @exception DHBmatrixAlgebra.DhbIllegalDimension if the vector
 *					and supplied vector do not have the same dimension.
 */
public DhbVector add ( DhbVector v) throws DhbIllegalDimension
{
		if ( this.dimension() != v.dimension() )
			throw new DhbIllegalDimension("Attempt to add a "
							+this.dimension()+"-dimension vector to a "
								+v.dimension()+"-dimension
vector");
```

```
        double[] newComponents = new double[ this.dimension()];
        for ( int i = 0; i < this.dimension(); i++)
                newComponents[i] = components[i] + v.components[i];
        return new DhbVector( newComponents);
}
/**
 * Sets all components of the receiver to 0.
 */
public void clear()
{
        for ( int i = 0; i < components.length; i++) components[i] = 0;
}
/**
 * @return double
 * @param n int
 */
public double component( int n)
{
        return components[n];
}
/**
 * Returns the dimension of the vector.
 * @return int
 */
public int dimension ( )
{
        return components.length;
}
/**
 * @return true if the supplied vector is equal to the receiver
 * @param v DHBmatrixAlgebra.DhbVector
 */
public boolean equals( DhbVector v)
{
        int n = this.dimension();
        if ( v.dimension() != n )
                return false;
        for ( int i = 0; i < n; i++)
        {
                if ( v.components[i] != components[i] )
                        return false;
        }
        return true;
}
/**
 * Computes the norm of a vector.
 */
public double norm ( )
{
        double sum = 0;
        for ( int i = 0; i < components.length; i++)
                sum += components[i]*components[i];
        return Math.sqrt( sum);
}
/**
 * @param x double
 */
public DhbVector normalizedBy ( double x )
{
		for ( int i = 0; i < this.dimension(); i++)
			components[i] /= x;
		return this;
}
/**
 * Computes the product of the vector by a number.
 * @return DHBmatrixAlgebra.DhbVector
 * @param d double
 */
public DhbVector product( double d)
{
		double newComponents[] = new double[components.length];
		for ( int i = 0; i < components.length; i++)
			newComponents[i] = d * components[i];
		return new DhbVector(newComponents);
}
/**
 * Compute the scalar product (or dot product) of two vectors.
 * @return double the scalar product of the receiver with the argument
 * @param v DHBmatrixAlgebra.DhbVector
 * @exception DHBmatrixAlgebra.DhbIllegalDimension if the dimension
 *
		of v is not the same.
 */
public double product ( DhbVector v) throws DhbIllegalDimension
{
		int n = v.dimension();
		if ( components.length != n )
			throw new DhbIllegalDimension(
						"Dot product with mismatched dimensions: "
						+components.length+", "+n);
		return secureProduct( v);
}
/**
 * Computes the product of the transposed vector with a matrix
 * @return MatrixAlgebra.DhbVector
 * @param a MatrixAlgebra.Matrix
 */
public DhbVector product ( Matrix a) throws DhbIllegalDimension
{
		int n = a.rows();
		int m = a.columns();
		if ( this.dimension() != n )
			throw new DhbIllegalDimension(
						"Product error: transposed of a "+this.dimension()
						+"-dimension vector cannot be multiplied with a "
										+n +" by
"+m+" matrix");
		return secureProduct( a);
}
/**
 * @param x double
 */
```

```
public DhbVector scaledBy ( double x )
{
        for ( int i = 0; i < this.dimension(); i++)
                components[i] *= x;
        return this;
}
/**
 * Compute the scalar product (or dot product) of two vectors.
 * No dimension checking is made.
 * @return double the scalar product of the receiver with the argument
 * @param v DHBmatrixAlgebra.DhbVector
 */
protected double secureProduct ( DhbVector v)
{
        double sum = 0;
        for ( int i = 0; i < v.dimension(); i++)
                sum += components[i]*v.components[i];
        return sum;
}
/**
 * Computes the product of the transposed vector with a matrix
 * @return MatrixAlgebra.DhbVector
 * @param a MatrixAlgebra.Matrix
 */
protected DhbVector secureProduct ( Matrix a)
{
        int n = a.rows();
        int m = a.columns();
        double[] vectorComponents = new double[m];
        for ( int j = 0; j < m; j++ )
        {
                vectorComponents[j] = 0;
                for ( int i = 0; i < n; i++)
                        vectorComponents[j] += components[i] * a.components[i][j];
        }
        return new DhbVector( vectorComponents);
}
/**
 * @return DHBmatrixAlgebra.DhbVector  subtract the supplied vector
 *
        to the receiver
 * @param v DHBmatrixAlgebra.DhbVector
 * @exception DHBmatrixAlgebra.DhbIllegalDimension if the vector
 * and supplied vector do not have the same dimension.
 */
public DhbVector subtract ( DhbVector v) throws DhbIllegalDimension
{
        if ( this.dimension() != v.dimension() )
                throw new DhbIllegalDimension("Attempt to add a "
                                                        +this.dimension()+"-dimension vector to a "
                                                                        +v.dimension()+"-dimension
vector");
        double[] newComponents = new double[ this.dimension()];
        for ( int i = 0; i < this.dimension(); i++)
                newComponents[i] = components[i] - v.components[i];
        return new DhbVector( newComponents);
}
/**
 * @return MatrixAlgebra.Matrix  tensor product with the specified
 *
                                              vector
 * @param v MatrixAlgebra.DhbVector      second vector to build tensor
 *
                          product with.
 */
public Matrix tensorProduct ( DhbVector v)
{
        int n = dimension();
        int m = v.dimension();
        double [][] newComponents = new double[n][m];
        for ( int i = 0; i < n; i++)
        {
                for ( int j = 0; j < m; j++)
                        newComponents[i][j] = components[i] * v.components[j];
        }
        return n == m ? new SymmetricMatrix( newComponents)
                                                      : new Matrix( newComponents);
}
/**
 * @return double[]         a copy of the components of the receiver.
 */
public double[] toComponents ( )
{
        int n = dimension();
        double[] answer = new double[ n];
        System.arraycopy( components, 0, answer, 0, n);
        return answer;
}
/**
 * Returns a string representation of the vector.
 * @return java.lang.String
 */
public String toString()
{
        StringBuffer sb = new StringBuffer();
        char[] separator = { '[', ' '};
        for ( int i = 0; i < components.length; i++)
        {
                sb.append( separator);
                sb.append( components[i]);
                separator[0] = ',';
        }
        sb.append(']');
        return sb.toString();
}
                                              }
```

Appendix K: DhbIllegalDimension

```
public class DhbIllegalDimension extends Exception {

/**
 * DhbIllegalDimension constructor comment.
 */
public DhbIllegalDimension() {
		super();
}
/**
 * DhbIllegalDimension constructor comment.
 * @param s java.lang.String
 */
public DhbIllegalDimension(String s) {
		super(s);
}
}
```

Appendix L: DhbNonSymmetricComponents

```
public class DhbNonSymmetricComponents extends Exception {

/**
 * DhbNonSymmetricComponents constructor comment.
 */
public DhbNonSymmetricComponents() {
		super();
}
/**
 * DhbNonSymmetricComponents constructor comment.
 * @param s java.lang.String
 */
public DhbNonSymmetricComponents(String s) {
		super(s);
}
}
```

Appendix M: LUPDecomposition

```
public class LUPDecomposition
{
/**
 * Rows of the system
 */
		private double[][] rows;
/**
 * Permutation
 */
		private int[] permutation = null;
/**
 * Permutation's parity
 */
		private int parity = 1;
/**
 * Constructor method
 * @param components double[][]
 * @exception DhbMatrixAlgebra.DhbIllegalDimension
 *                                                          the supplied matrix is not
square
 */
public LUPDecomposition ( double[][]components)
                                                                                  throws
DhbIllegalDimension
{
		int n = components.length;
		if ( components[0].length != n )
			throw new DhbIllegalDimension("Illegal system: a"+n+" by "
				+components[0].length+" matrix is not a square matrix");
		rows = components;
		initialize();
}
/**
 * Constructor method.
 * @param m DhbMatrixAlgebra.Matrix
 * @exception DhbMatrixAlgebra.DhbIllegalDimension
 *                                                          the supplied matrix is not
square
 */
public LUPDecomposition ( Matrix m) throws DhbIllegalDimension
{
		if ( !m.isSquare() )
			throw new DhbIllegalDimension(
						"Supplied matrix is not a square matrix");
		initialize( m.components);
}
/**
 * Constructor method.
 * @param m DhbMatrixAlgebra.DhbSymmetricMatrix
 */
public LUPDecomposition ( SymmetricMatrix m)
```

```
{
        initialize( m.components);
}
/**
 * @return double[]
 * @param xTilde double[]
 */
private double[] backwardSubstitution( double[] xTilde)
{
        int n = rows.length;
        double[] answer = new double[n];
        for ( int i = n - 1; i >= 0; i--)
        {
                answer[i] = xTilde[i];
                for ( int j = i + 1; j < n; j++)
                        answer[i] -= rows[i][j] * answer[j];
                answer[i] /= rows[i][i];
        }
        return answer;
}
private void decompose()
{
        int n = rows.length;
        permutation = new int[n];
        for ( int i = 0; i < n; i++ )
                permutation[i] = i;
        parity = 1;
        try {
                        for ( int i = 0; i < n; i++)
                        {
                                swapRows( i, largestPivot( i));
                                pivot( i);
                        }
                } catch ( ArithmeticException e) { parity = 0;};
}
/**
 * @return boolean          true if decomposition was done already
 */
private boolean decomposed()
{
        if ( parity == 1 && permutation == null )
                decompose();
        return parity != 0;
}
/**
 * @return double[]
 * @param c double[]
 */
public double determinant()
{
        if ( !decomposed() )
                return Double.NaN;
        double determinant = parity;
        for ( int i = 0; i < rows.length; i++ )
                determinant *= rows[i][i];
        return determinant;
```

```
}
/**
 * @return double[]
 * @param c double[]
 */
private double[] forwardSubstitution( double[] c)
{
            int n = rows.length;
            double[] answer = new double[n];
            for ( int i = 0; i < n; i++)
            {
                        answer[i] = c[permutation[i]];
                        for ( int j = 0; j <= i - 1; j++)
                                    answer[i] -= rows[i][j] * answer[j];
            }
            return answer;
}
private void initialize ()
{
            permutation= null;
            parity = 1;
}
/**
 * @param components double[][]  components obtained from constructor methods.
 */
private void initialize ( double[][] components)
{
            int n = components.length;
            rows = new double[n][n];
            for ( int i = 0; i < n; i++)
            {
                        for ( int j = 0; j < n; j++)
                                    rows[i][j] = components[i][j];
            }
            initialize();
}
/**
 * @return double[]
 * @param c double[]
 */
public double[][] inverseMatrixComponents()
{
            if ( !decomposed() )
                        return null;
            int n = rows.length;
            double[][] inverseRows = new double[n][n];
            double[] column = new double[n];
            for ( int i = 0; i < n; i ++)
            {
                        for ( int j = 0; j < n; j++ )
                                    column[j] = 0;
                        column[i] = 1;
                        column = solve( column);
                        for ( int j = 0; j < n; j++ )
                                    inverseRows[i][j] = column[j];
            }
```

```
        return inverseRows;
}
/**
 * @return int
 * @param k int
 */
private int largestPivot(int k)
{
        double maximum = Math.abs( rows[k][k]);
        double abs;
        int index = k;
        for ( int i = k + 1; i < rows.length; i++)
        {
                abs = Math.abs( rows[i][k]);
                if ( abs > maximum )
                {
                        maximum = abs;
                        index = i;
                }
        }
        return index;
}
/**
* @param k int
 */
private void pivot( int k)
{
        double inversePivot = 1 / rows[k][k];
        int k1 = k + 1;
        int n = rows.length;
        for ( int i = k1; i < n; i++)
        {
                rows[i][k] *= inversePivot;
                for ( int j = k1; j < n; j++)
                        rows[i][j] -= rows[i][k] * rows[k][j];
        }
}
/**
 * @return double[]
 * @param c double[]
 */
public double[] solve( double[] c)
{
        return decomposed()
                                        ? backwardSubstitution( forwardSubstitution( c))
                                        : null;
}
/**
 * @return double[]
 * @param c double[]
 */
public DhbVector solve( DhbVector c)
{
        double[] components = solve( c.components);
        if ( components == null )
                return null;
```

```
		return components == null ? null : new DhbVector( components);
}
/**
 * @param i int
 * @param k int
 */
private void swapRows( int i, int k)
{
		if ( i != k )
		{
				double temp;
				for ( int j = 0; j < rows.length; j++ )
				{
						temp = rows[i][j];
						rows[i][j] = rows[k][j];
						rows[k][j] = temp;
				}
				int nTemp;
				nTemp = permutation[i];
				permutation[i] = permutation[k];
				permutation[k] = nTemp;
				parity = -parity;
		}
}
/**
 * Make sure the supplied matrix components are those of
 * a symmetric matrix
 * @param components double
 */
public static void symmetrizeComponents( double[][] components)
{
		for ( int i = 0; i < components.length; i++ )
				{
						for ( int j = i + 1; j < components.length; j++ )
						{
								components[i][j] += components[j][i];
								components[i][j] *= 0.5;
								components[j][i] = components[i][j];
						}
				}
}
/**
 * Returns a String that represents the value of this object.
 * @return a string representation of the receiver
 */
public String toString()
{
		StringBuffer sb = new StringBuffer();
		char[] separator = { '[', ' '};
		int n = rows.length;
		for ( int i = 0; i < n; i++)
		{
				separator[0] = '{';
				for ( int j = 0; j < n; j++)
				{
						sb.append( separator);
```

```
                    sb.append( rows[i][j]);
                    separator[0] = ' ';
                }
                sb.append('}');
                sb.append('\n');
            }
            if ( permutation != null )
            {
                sb.append( parity == 1 ? '+' : '-');
                sb.append("( " + permutation[0]);
                for ( int i = 1; i < n; i++)
                    sb.append(", " + permutation[i]);
                sb.append(')');
                sb.append('\n');
            }
            return sb.toString();
        }
    }
```