# ASEN 4057
# Final Project

Alex Bertman, Aaron McCusker, Jeremy Muesing

May 6, 2017

## 1. Description of C Program to Improve

The original coding proposal was to test the merits of buying stocks for investment purposes based solely on past performance of the stock. This likely dubious approach to investing was be tested by analyzing end-of-day stock data from publicly traded US companies over the past 37 years. We checked the risk of each stock every month by calculating the stock's Beta value (stock's movement compared to the overall market). The equation to find Beta can be seen in Equation 1. We then calculated the overall trend of the stock over a period of time, and made a buy / don't buy decision based on the trend and Beta value. We then checked the performance of the stock after the buy / don't buy decision using the data from the time period following the decision. By performing this multiple times per stock for every stock in the database, we wanted to make a statement regarding the validity of purchasing based purely on a stock's past performance.

$$Beta = \frac{Covariance(StockA, Market)}{Variance(StockA)} \tag{1}$$

The validity of the stock purchase was analyzed by finding the "Alpha" value from the stock. The Alpha value is a measurement of whether or not the investment performed well compared to the market, and if the risk that was taken when buying the stock was worth it. Equation 2 shows how Alpha is found where R is the return. Any positive Alpha value is good, but highly regarded investment banks claim to have an average Alpha value of 4 per trade. Note that a positive Alpha value does not indicate that the investor made money, but instead shows whether the investor performed well compared to the rest of the market.

$$Alpha = R_{Realized} - (R_{RiskFree} + (R_{Market} - R_{RiskFree}) * Beta) \tag{2}$$

The large quantity of data (hundreds of thousands of data points) must be extracted from downloaded files (from Quandl.com) and then must be analyzed. The large volume of data that must be dealt with leaves plenty of room for optimization, and the fact that the tasks associated with each stock are very similar, we believed we could effectively employ parallel computing to reach a conclusion regarding our investment strategy in a relatively quick manner.

# 2. Serial Code Optimization

## 2.1 Profile Report and Discussion of Performance of Original C Code

The original C Code did not efficiently perform operations. The basic outline allowed the history of a single stock to be read in, but the S&P500 was also read in for each stock to compare to. The original code also computed values for each day. This related to both the Beta values and Alpha values even though we only needed a small portion of these for our results.

The overall execution time for the original C code was 381.329 milliseconds. Though this is fast, the team feels the runtime can be greatly reduced.

Using *gprof* to profile the code yielded poor results. *gprof* does not include time spent in standard libraries in its timing, and since most of the time the code is either scanning or printing using a standard library function, almost none of the code's overall runtime is captured. However, *gprof* did yield positive results for the Beta and Alpha functions which did not require any scanning or printing.

The run time for each Alpha and Beta function was less than 1 $\mu s$ each. Within those functions, it was found that the most time-consuming process was actually returning the Alpha and Beta value to the main function. The Beta function was called 158432 times, so the time spent returning the value began to add up. In fact, almost half of the code's run time (outside of scanning and printing) was spent returning the Beta value to the main function. On the other hand, the Alpha function was only called 3839 times. Because it was only called 1/42nd as often as the Beta function, the returns from the Alpha function did not contribute significantly to the code's runtime.

It was noted above that almost half of the runtime for the code (aside from scanning and printing) is taken by returning the Beta value to the main function. Almost the entire other half of the non-scanning or printing runtime is taken up by an *if statement* within the reading function. The statement checks whether or not the company's stock split on the given day. Though it is a simple check, the check is performed over 150,000 times, so it really begins to add up. Still, the time spent returning Beta and the time spent checking if the stock split combine for less than 10 milliseconds of the 381.329 millisecond runtime.

```
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00   158432     0.00     0.00  beta
  0.00      0.00     0.00     3839     0.00     0.00  alpha
  0.00      0.00     0.00       37     0.00     0.00  ReadFile

 %          the percentage of the total running time of the
time        program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self       the number of seconds accounted for by this
seconds     function alone.  This is the major sort for this
            listing.

calls       the number of times this function was invoked, if
            this function is profiled, else blank.

 self       the average number of milliseconds spent in this
ms/call     function per call, if this function is profiled,
           else blank.

 total      the average number of milliseconds spent in this
ms/call     function and its descendents per call, if this
           function is profiled, else blank.

name        the name of the function.  This is the minor sort
            for this listing. The index shows the location of
           the function in the gprof listing. If the index is
           in parenthesis it shows where it would appear in
           the gprof listing if it were to be printed.
Copyright (C) 2012 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.
                    Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) no time propagated

index % time    self  children    called     name
                0.00    0.00    3875/158432      beta [2]
                0.00    0.00  154557/158432      main [10]
[1]      0.0    0.00    0.00  158432         ReadFile [1]
-----------------------------------------------
                0.00    0.00    3839/3839        main [10]
[2]      0.0    0.00    0.00    3839         beta [2]
                0.00    0.00    3875/158432      ReadFile [1]
-----------------------------------------------
                0.00    0.00      37/37          main [10]
```

```
[3]      0.0    0.00    0.00      37         alpha [3]
-----------------------------------------------
                                  42             main [10]
[10]     0.0    0.00    0.00       0+42      main [10]
                0.00    0.00  154557/158432      ReadFile [1]
                0.00    0.00    3839/3839        beta [2]
                0.00    0.00      37/37          alpha [3]
                                  42             main [10]
-----------------------------------------------
```

This table describes the call tree of the program, and was sorted by the total amount of time spent in each function and its children.

Each entry in this table consists of several lines. The line with the index number at the left hand margin lists the current function. The lines above it list the functions that called this function, and the lines below it list the functions this one called. This line lists:

- index — A unique number given to each element of the table. Index numbers are sorted numerically. The index number is printed next to every function name so it is easier to look up where the function is in the table.

- % time — This is the percentage of the `total' time that was spent in this function and its children. Note that due to different viewpoints, functions excluded by options, etc, these numbers will NOT add up to 100%.

- self — This is the total amount of time spent in this function.

- children — This is the total amount of time propagated into this function by its children.

- called — This is the number of times the function was called. If the function called itself recursively, the number only includes non-recursive calls, and is followed by a `+' and the number of recursive calls.

- name — The name of the current function. The index number is printed after it. If the function is a member of a cycle, the cycle number is printed between the function's name and the index number.

For the function's parents, the fields have the following meanings:

- self — This is the amount of time that was propagated directly from the function into this parent.

- children — This is the amount of time that was propagated from the function's children into this parent.

- called — This is the number of times this parent called the function `/' the total number of times the function was called. Recursive calls to the function are not included in the number after the `/'.

```
    name      This is the name of the parent.  The parent's index
              number is printed after it.  If the parent is a
              member of a cycle, the cycle number is printed between
              the name and the index number.

 If the parents of the function cannot be determined, the word
 `<spontaneous>' is printed in the `name' field, and all the other
 fields are blank.

 For the function's children, the fields have the following meanings:

    self      This is the amount of time that was propagated directly
              from the child into the function.

    children  This is the amount of time that was propagated from the
              child's children to the function.

    called    This is the number of times the function called
              this child `/' the total number of times the child
              was called.  Recursive calls by the child are not
              listed in the number after the `/'.

    name      This is the name of the child.  The child's index
              number is printed after it.  If the child is a
              member of a cycle, the cycle number is printed
              between the name and the index number.

 If there are any cycles (circles) in the call graph, there is an
 entry for the cycle-as-a-whole.  This entry shows who called the
 cycle (as parents) and the members of the cycle (as children.)
 The `+' recursive calls entry shows the number of function calls that
 were internal to the cycle, and the calls entry for each member shows,
 for that member, how many times it was called from other members of
 the cycle.
Copyright (C) 2012 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.
Index by function name

   [1] ReadFile                [3] alpha                   [2] beta
```

### 2.2 Proposed Serial Code Improvements

Our proposed improvements followed the ”Improving the operation count” as stated in the assignment. We do this by only reading in the S&P500 once, seeing as all stocks are compared to the same market stock for each instance. The second improvement is to only perform calculations for the required time interval. This is a variable set in the code, however providing an interval means there is a considerable amount of data that could be ignored for computation. This extends further to the Alpha value. Based on the Beta value found at each time interval, the

Alpha was was only computed for instances we determined to ”buy” situations. The Alpha value is a measure of the performance of that purchase. Previously the alpha value was being calculated even in ”non-buy” situations.

### 2.3 Timing Comparison of Original C Code and Optimized C Code

The timing comparison between the original C code and the optimized code can be seen below in Table 1.

Table 1: Timing Comparison of Original C Code and Optimized C Code

| Code | Runtime (ms) |
|---|---|
| Original C Code | 381.33 |
| Optimized C Code | 184.52 |

### 2.4 Profile Report and Discussion of Performance of Optimized C Code

The optimized C code runs very quickly and efficiently. It analyzes almost four decades worth of stock data for 46 stocks in under one second. The total time that the code takes to run is easy to measure using the *time.h* standard library. However, analyzing the time using *gprof* has proven difficult. Some research showed that *gprof* does not measure time spent in standard libraries. This is a big issue because the bulk of the computation time for the code is spent scanning and printing using standard library functions. Because of this issue, *gprof* does not display an accurate timing breakdown.

One valuable aspect obtained from the *gprof* profile is that there is one line in a single function that takes up over 50% of non-scanning or printing processing time. An *if statement* in the function *ReadFile.c* checks to see if the stock has split. This check must be performed for every single day read in by the function. That *if statement* is extremely time consuming as it must be run thousands of times for each stock that is read in. If the team did not have to read or write any data and was just performing analysis on pre-read data, that *if statement* would be the most time consuming aspect of the code.

```
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00   158432     0.00     0.00  __libc_csu_init
  0.00      0.00     0.00     3876     0.00     0.00  _fini
  0.00      0.00     0.00       42     0.00     0.00  Output

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
          else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
          function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
          the function in the gprof listing. If the index is
          in parenthesis it shows where it would appear in
          the gprof listing if it were to be printed.
Copyright (C) 2012 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.
                    Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) no time propagated

index % time    self  children    called     name
                0.00    0.00      42/42          ReadFile [2]
[1]      0.0    0.00    0.00      42         Output [1]
                0.00    0.00  154557/158432      __libc_csu_init [12]
-----------------------------------------------
                0.00    0.00    3875/158432      etext [8]
                0.00    0.00  154557/158432      Output [1]
[12]     0.0    0.00    0.00  158432         __libc_csu_init [12]
-----------------------------------------------
                0.00    0.00      37/3876        beta [5]
```

```
                0.00    0.00    3839/3876        alpha [3]
[13]     0.0    0.00    0.00    3876          _fini [13]
-----------------------------------------------
```

This table describes the call tree of the program, and was sorted by
the total amount of time spent in each function and its children.

Each entry in this table consists of several lines.  The line with the
index number at the left hand margin lists the current function.
The lines above it list the functions that called this function,
and the lines below it list the functions this one called.
This line lists:

- index — A unique number given to each element of the table.
  Index numbers are sorted numerically.
  The index number is printed next to every function name so
  it is easier to look up where the function is in the table.

- % time — This is the percentage of the `total' time that was spent
  in this function and its children.  Note that due to
  different viewpoints, functions excluded by options, etc,
  these numbers will NOT add up to 100%.

- self — This is the total amount of time spent in this function.

- children — This is the total amount of time propagated into this
  function by its children.

- called — This is the number of times the function was called.
  If the function called itself recursively, the number
  only includes non-recursive calls, and is followed by
  a `+' and the number of recursive calls.

- name — The name of the current function.  The index number is
  printed after it.  If the function is a member of a
  cycle, the cycle number is printed between the
  function's name and the index number.

For the function's parents, the fields have the following meanings:

- self — This is the amount of time that was propagated directly
  from the function into this parent.

- children — This is the amount of time that was propagated from
  the function's children into this parent.

- called — This is the number of times this parent called the
  function `/' the total number of times the function
  was called.  Recursive calls to the function are not
  included in the number after the `/'.

- name — This is the name of the parent.  The parent's index
  number is printed after it.  If the parent is a
  member of a cycle, the cycle number is printed between
  the name and the index number.

If the parents of the function cannot be determined, the word

```
`<spontaneous>' is printed in the `name' field, and all the other
fields are blank.

For the function's children, the fields have the following meanings:

    self     This is the amount of time that was propagated directly
             from the child into the function.

    children This is the amount of time that was propagated from the
             child's children to the function.

    called   This is the number of times the function called
             this child `/' the total number of times the child
             was called.  Recursive calls by the child are not
             listed in the number after the `/'.

    name     This is the name of the child.  The child's index
             number is printed after it.  If the child is a
             member of a cycle, the cycle number is printed
             between the name and the index number.

 If there are any cycles (circles) in the call graph, there is an
 entry for the cycle-as-a-whole.  This entry shows who called the
 cycle (as parents) and the members of the cycle (as children.)
 The `+' recursive calls entry shows the number of function calls that
 were internal to the cycle, and the calls entry for each member shows,
 for that member, how many times it was called from other members of
 the cycle.
Copyright (C) 2012 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.
Index by function name

   [1] Output                  [12] __libc_csu_init        [13] _fini
```

# 3. Parallelization of Optimized Code

## 3.1 Proposed Strategy for Code Parallelization

The strategy for parallelization was to run an entire stock on a single process and share the S&P500 between each process. With 45 different stocks being analyzed over 37 years, we concluded this would be a reasonable split per core so that no core would be waiting for another to finish. Each stock contains a comparable (but not identical) number of computations.

This approach removes almost all serial processes from the code. The only serial process is the reading in of the S&P500 data at the beginning of the code.

### 3.2 Scalability Study on Virtual Machine with up to Eight Processes

The scalability for our code, as one can see in Table 2, is not very good. In fact, the code gets slower as the number of processes increases. It should be noted that the time for the serial portion of the code to run is roughly 10 ms.

Table 2: Runtimes for various process numbers

| Processes | Runtime (ms) |
|---|---|
| 1 | 184.51 |
| 2 | 194.05 |
| 4 | 211.18 |
| 8 | 382.73 |

### 3.3 Profile Report and Discussion of Performance on Virtual Machine

Surprisingly, the code performed slower in parallelization. It was discovered that scanning and printing, which are the dominant tasks within the code, do not parallelize well. In fact, they actually slow down the process compared to performing the entire code in serial.

The team attempted to create a *gprof* profile for the 8-process execution. However, once again, *gprof* had enormous issues due to the fact that it was not measuring the time spent in standard libraries. Therefore, the profile report shows absolutely nothing of worth. It claims that the code runs in less than 10 ms (which is very incorrect) and it thinks that the slowest functions are Alpha and Beta, which is also very untrue. The runtime percentage breakdown for each function is identical to the percentage breakdown running in only serial (that is, the statement checking whether or not the code split and the returning of Beta take up almost all of the non-scanning or printing time). *gprof* showed that both Alpha and Beta functions ran for less than 10 milliseconds total (*gprof* does not show significant figures more precise than 10 milliseconds).

```
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00   158432     0.00     0.00  beta
  0.00      0.00     0.00     3839     0.00     0.00  alpha
  0.00      0.00     0.00       42     0.00     0.00  ReadFile
  0.00      0.00     0.00       37     0.00     0.00  Output

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
ms/call    function per call, if this function is profiled,
          else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
          function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
          the function in the gprof listing. If the index is
          in parenthesis it shows where it would appear in
          the gprof listing if it were to be printed.
Copyright (C) 2012 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.
                  Call graph (explanation follows)


granularity: each sample hit covers 2 byte(s) no time propagated

index % time    self  children    called     name
                0.00    0.00    3875/158432      alpha [2]
                0.00    0.00  154557/158432      frame_dummy [9]
[1]      0.0    0.00    0.00  158432         beta [1]
-----------------------------------------------
                0.00    0.00    3839/3839        frame_dummy [9]
[2]      0.0    0.00    0.00    3839         alpha [2]
                0.00    0.00    3875/158432      beta [1]
-----------------------------------------------
```

```
                0.00    0.00       42/42          frame_dummy [9]
[3]      0.0    0.00    0.00       42          ReadFile [3]
-----------------------------------------------
                0.00    0.00       37/37          frame_dummy [9]
[4]      0.0    0.00    0.00       37          Output [4]
-----------------------------------------------
```

This table describes the call tree of the program, and was sorted by
the total amount of time spent in each function and its children.

Each entry in this table consists of several lines.  The line with the
index number at the left hand margin lists the current function.
The lines above it list the functions that called this function,
and the lines below it list the functions this one called.
This line lists:

- **index** A unique number given to each element of the table.
  Index numbers are sorted numerically.
  The index number is printed next to every function name so
  it is easier to look up where the function is in the table.

- **% time** This is the percentage of the `total' time that was spent
  in this function and its children.  Note that due to
  different viewpoints, functions excluded by options, etc,
  these numbers will NOT add up to 100%.

- **self** This is the total amount of time spent in this function.

- **children** This is the total amount of time propagated into this
  function by its children.

- **called** This is the number of times the function was called.
  If the function called itself recursively, the number
  only includes non-recursive calls, and is followed by
  a `+' and the number of recursive calls.

- **name** The name of the current function.  The index number is
  printed after it.  If the function is a member of a
  cycle, the cycle number is printed between the
  function's name and the index number.

For the function's parents, the fields have the following meanings:

- **self** This is the amount of time that was propagated directly
  from the function into this parent.

- **children** This is the amount of time that was propagated from
  the function's children into this parent.

- **called** This is the number of times this parent called the
  function `/' the total number of times the function
  was called.  Recursive calls to the function are not
  included in the number after the `/'.

- **name** This is the name of the parent.  The parent's index
  number is printed after it.  If the parent is a
  member of a cycle, the cycle number is printed between

```
             the name and the index number.

 If the parents of the function cannot be determined, the word
 `<spontaneous>' is printed in the `name' field, and all the other
 fields are blank.

 For the function's children, the fields have the following meanings:

     self     This is the amount of time that was propagated directly
              from the child into the function.

     children This is the amount of time that was propagated from the
              child's children to the function.

     called   This is the number of times the function called
              this child `/' the total number of times the child
              was called.  Recursive calls by the child are not
              listed in the number after the `/'.

     name     This is the name of the child.  The child's index
              number is printed after it.  If the child is a
              member of a cycle, the cycle number is printed
              between the name and the index number.

 If there are any cycles (circles) in the call graph, there is an
 entry for the cycle-as-a-whole.  This entry shows who called the
 cycle (as parents) and the members of the cycle (as children.)
 The `+' recursive calls entry shows the number of function calls that
 were internal to the cycle, and the calls entry for each member shows,
 for that member, how many times it was called from other members of
 the cycle.
Copyright (C) 2012 Free Software Foundation, Inc.

Copying and distribution of this file, with or without modification,
are permitted in any medium without royalty provided the copyright
notice and this notice are preserved.
Index by function name

   [4] Output                  [2] alpha
   [3] ReadFile                [1] beta
```

## 4. Summary of Findings and Potential Future Improvements

There are two major findings presented below. The first is the results of the algorithm developed for this project.

Based off of 1 month intervals, ”buy” and ”no-buy” decisions were made on 45 stocks over 37 years. The Beta values of each stock are shown in Figure 1. This is a measure of volatility of a stock compared to the market. There are a few key spots to point out such as the housing bubble. This produced highly volatile stocks in the negative direction.

If a stock was trending upwards in the last month, and its absolute value of Beta was greater than 1, the stock

was purchased.

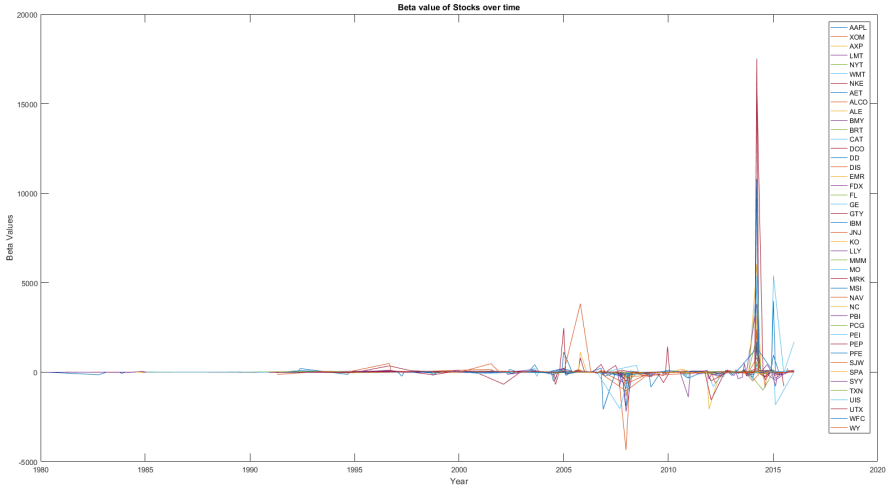


Figure 1: Beta value of all Stocks over 37 years

The next value to compute was the Alpha value. This is ultimately a measure of hindsight. Did the stock outperform or under-perform the market, and was the risk of purchase worth it? In Figure 2 one can see some interesting trends such as Apple when Steve Jobs was fired in the mid 80's. One can also see how the market reacted to the housing bubble as well as more recent changes in recent years.

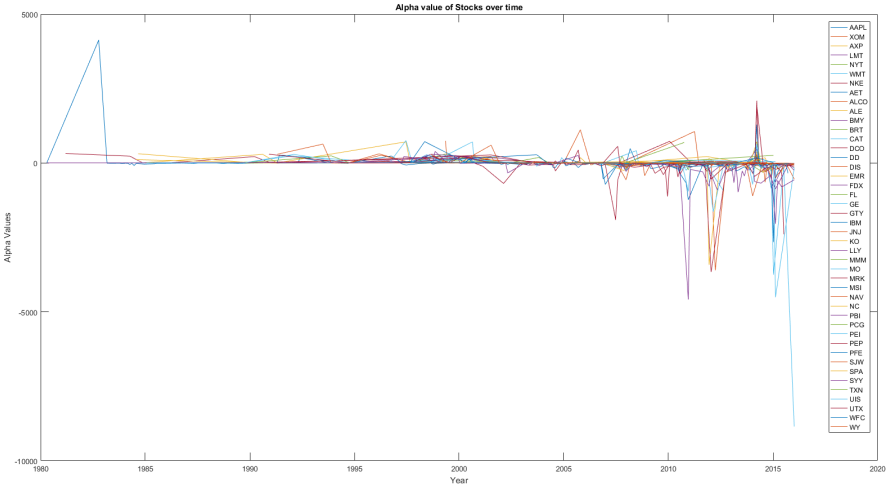


Figure 2: Alpha value of all Stocks over 37 years

The final graph shown below is the total alpha value accumulated over time. One can see once again the the algorithm performs well for a given time up until the 2009 crash. At this point, several stocks start rapidly under-performing and losing money.

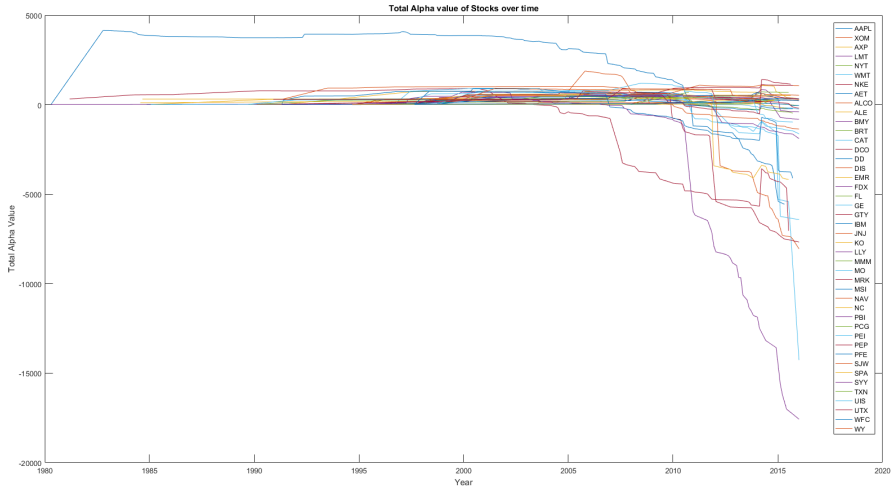


Figure 3: Total Alpha value of all Stocks over 37 years

Overall, the strategy of purchasing stocks based on nothing but past performance is not advised. The stocks that we chose we subject to major selection bias as they were all companies that were publically traded in 1980 and are still publically traded today. None of the companies that we analyzed went bankrupt, so we never risked losing everything when purchasing stocks. Additionally, the strategy only made money when the market was steadily growing. When the market becomes turbulent, all of the positives that the strategy has go out the window.

Future work may include incorporating most decision points to the buy / no-buy decision. Decision points may include things such as dividends paid, which is something that Apple does extensively but which is not covered in our current code. Additionally, the code required nicely groomed data. The raw data from Quandl.com was too difficult and variable to read in, so we nicely groomed the data before feeding it into the code. The code could become much more useful and robust if it were able to read in the default data style from Quandl.com.