MTH 4600 - Project 6
Portfolio Optimization via Metropolis

Martin Bachurin
Anton Gurshumov
John Hemminger

Baruch College

April 21, 2023

For problem 1, suppose $x \neq y$ are two distinct portfolios that are permitted to have both long and short positions. Then $p = \lambda x + (1-\lambda)y$ is another portfolio for any real number λ. We first show that $Var(p)$ is quadratic in $\lambda$of the form $a\lambda^2 + b\lambda + c$ for some real numbers a, b, and c with $a > 0$.

You may use that $\mathbf{V}$ is PD to establish that $a > 0$. Calculate a in terms of x, y, and $Var(*)$.

Use this to argue that $Var(x)$ cannot have multiple local minima as x ranges over all portfolios.

What does that say about the best temperature parameter for problems 2 and 3?

$$\begin{aligned}
Var(p) &= a\lambda^2 + b\lambda + c \\
Var(p) &= Var(\lambda x + (1-\lambda)y) = Var(\lambda x) + Var(1-\lambda)y + 2Cov(\lambda x, (1-\lambda)y) \\
Var(p) &= \lambda^2 Var(x) + (1-\lambda)^2 Var(y) + 2\lambda(1-\lambda)Cov(x,y) \\
Var(p) &= \lambda^2 Var(x) + Var(y) - 2\lambda Var(y) + \lambda^2 Var(y) + 2\lambda Cov(x,y) - 2\lambda^2 Cov(x,y) \\
Var(p) &= \lambda^2(Var(x) + Var(y) - 2Cov(x,y)) + 2\lambda(Cov(x,y) - Var(y)) + Var(y) \\
a &= Var(x) + Var(y) - 2Cov(x,y) = Var(x-y)
\end{aligned}$$

Now we use the fact that V is PD to establish $a > 0$.

$$Var(x-y) = (x-y)^T \mathbf{V}(x-y)$$

Since V is a positive definite matrix, we know that by definition, $x^T Ax$ is always positive. Therefore we have that $a > 0$.

By the above argument, we have established that $Var(p)$ is strictly a positive quadratic, and therefore any portfolio x will be convex as x ranges over all portfolios. This implies that x cannot have multiple local minima by its geometry. This is true since p contains x since p is a linear combination of all portfolios. This implies that the best temperature parameter is 0 since there are no stable states due to the energy being quadratic in nature.

For problem 2, we find the minimum variance portfolio when both long and short positions are allowed for individual stocks. We will find the allocation to each stock to the nearest 0.01 penny. First, to implement Metropolis, we must define the notion of state space, neighboring states, energy, and temperature. The state space for this problem is the set of all portfolios such that their components sum to 100. This is an infinite set of states; however, we make the set countable by limiting the possible weights to values incremented by 0.01. The neighboring states can be defined by the portfolios that are attainable by choosing one stock to increase by 0.01 and choosing a different stock to decrease by 0.01. By this definition, we can find the number of neighboring states by the fact that the order of buying/selling is relevant; therefore, we can apply 50 permutation 2, which gives 50 x 49. The definition of energy is trivial. We will take it to be the variance of a portfolio. Finally, we will choose temperatures close to 0 by the logic established in question 1.

Implementation: The algorithm works in 5 stages. First, we get the data needed for the matrix **V**, then we allocate the memory, next initialize a random portfolio, then we run the metropolis algorithm, and finally, we delete the memory. To initialize the portfolio, we continuously choose an allocation for each of the stocks by generating a uniform on the interval -1 to 1, multiplying this number by 10000, flooring it, and then dividing the number by 100. This ensures that we get a value to two decimals. As we allocate these values, we update the remaining allocation needed by subtracting 100 from the total amount allocated in the final step. This ensures a random portfolio with a total value of 100. Next, we run the Metropolis function, which requires a temperature parameter. We found that the algorithm works best when the temperature is very close to 0. We first measure the energy of the initial portfolio and start running 100,000,000 simulations of the Markov chain. This begins with generating a proposed neighbor (incrementing two random stocks by +/- 0.01), measuring its energy (variance calculation), and then deciding whether this portfolio has a lower energy than the lowest neighbor found. We do this by first calculating ΔE and checking if the change is negative. If so, we will obviously accept. If not, we will accept if the proposed probability transition is greater than or equal to a freshly generated uniform. In this step, we calculate $p = e^{-\Delta E/T}$. For this problem, since T is very small, p is very close to 0, so we will reject this probability almost surely. Finally, if we have accepted the transition, we will update the energy of the current portfolio and the newly found minimum with its associated portfolio. If we do not accept this neighboring state, we simply generate a new neighbor, as explained before, and repeat the iteration until we have done so 10,000,000 times. We can compare our results to the theoretical minimum computed by $\frac{\vec{e}^T \mathbf{V}}{\vec{e}^T \mathbf{V} \vec{e}}$. Where $\vec{e} = \{1, 1, ..., 1, 1\}$. We ran the comparison test and confirmed that the results look nearly identical, as seen in the code.

MVP with No Constraints	Min Var = 0.454419	MVP with No Constraints	Min Var = 0.454419
Stocks (1-25)	Allocation($)	Stocks (26-50)	Allocation($)
MS	14.72	PCLN	-12.82
RTN	24.69	HD	-7.88
OXY	16.65	MDT	7.48
TWX	-2.16	PFE	-35.75
F	16.31	CAT	-16.65
AIG	-0.7	JNJ	-51.06
ACN	-22.51	GOOGL	-11.99
HON	8.6	CELG	-2.93
QCOM	26.04	NEE	30.42
BA	-17.94	DD	-.06
UNH	-4.86	WFC	-0.4
PG	-2.53	KHC	1.22
HAL	20.51	CVX	4.52
INTC	-8.48	MMM	9.59
BLK	-13.25	MSFT	0.18
UPS	4.26	FDX	13.46
UNP	-9.19	VZ	24.81
V	17.34	KO	12.35
NKE	8.23	TGT	24.5
BIIB	6.38	WBA	-14.63
SLB	-16.56	MCD	15.03
COST	-4.16	CVS	-13.65
SPG	12.95	KMI	-4.21
MO	6.65	LLY	31.12
AMZN	8.06	GD	8.3

For problem 3, we will repeat problem two with the constraint of not allowing any short positions for individual stocks. The definitions of the Metropolis algorithm stay the same; however, to implement the constraint, we will simply force the energy of states with negative components to have high energy, for example, 1000. This is not a significant change in terms of implementation, however, we would expect the minimum variance of this portfolio to be higher than a portfolio with no constraints since we do not have as much freedom as before. We repeat everything as before with the change described above, and this results in the following.

MVP with No Short Selling	Min Var = 4.02856
Stocks	Allocation($)
RTN	7.86
OXY	8.03
F	3.59
UNH	4.15
PG	1.95
UNP	4.36
NKE	10.27
BIIB	1.37
NEE	14.08
WFC	1.85
KHC	2.44
MSFT	2.5
FDX	4.54
VZ	12.05
TGT	3.31
KMI	1.5
LLY	15.57
GD	0.58

For Problem 4, we will call a portfolio of these 50 stocks ‘simple’ if all the stocks present in the portfolio have equal weight and sum to \$100. Here, we must change the definition of the state space and the notion of neighbors. In this example, the state space will be the set of all portfolios such that there are 10 or more stocks in the portfolio, and the weights are the same amongst all chosen stocks. In general, the neighboring states are portfolios with 10 or more stocks such that we can change the allocation, either buying/selling an extra stock and redistributing \$100 to the newly proposed allocation of stocks. More specifically, we can choose a position in the portfolio, and if it is non-zero, we will set it to zero and redistribute the chosen value amongst the stocks that are non-zero equally. Otherwise, if the chosen position is zero, we will set it equal to 100 divided by the number of non-zero elements plus 1 and reset the remaining non-zero elements to this value as well. As before, we wish to find the portfolio with the lease variance. As for the expectation of the results, we expect the minimum variance to be greater than previously seen portfolios since the conditions have become even more constrained.

When implementing the code for the proposed neighbor, we must keep in mind the boundary conditions that must be met. First, we check if we are in a state with 40 zeros since we must add a stock randomly to the portfolio and redistribute the weights accordingly before measuring its energy. The same treatment is made with 0 zeros, where we must add a zero randomly and redistribute the values accordingly. If we are not in a boundary case, then we will randomly choose a neighboring state and add or subtract a zero based on a coin flip. Another consideration that must be made in this case is the correctly chosen temperature. We do this empirically by running the program repeatedly with various temperatures. Once we see that the variance starts to increase after decreasing in previous iterations, we stop, and we are left with our temperature parameter of 0.4.

MVP for 'Simple' portfolios	Min Var = 4.45384
Stocks	Allocation($)
RTN	9.09
OXY	9.09
UNH	9.09
PG	9.09
UNP	9.09
NKE	9.09
NEE	9.09
KHC	9.09
FDX	9.09
VZ	9.09
LLY	9.09

For question 5, we find the non-ground stable states for the simple portfolios. We first define a state x as stable if $E(y) > E(x)$ for all neighbors y of x. In the context of question 4, a non-ground stable state x is one whose energy $E(x)$ is not minimal; $E(x) > E_0$. In order to achieve this, we ran our algorithm with zero-temperature dynamics and monitored where the energy stopped decreasing by visiting its neighbors and started rejecting all future proposals. Then we verified the stability by computing the energy of its 50 neighbors. In the code, we track the number of rejections of future components, and once that number hits 1000, we can say with confidence that it is stuck. After testing a few different seeds, we have the following results.

Var. of Neighbors	Stable State Var. = 4.53029
Neighbors of Stable State (1-25)	Variance
1	5.36642
2	4.60944
3	4.82776
4	4.83177
5	4.63869
6	4.91229
7	4.94465
8	4.63456
9	4.77083
10	4.98508
11	4.56828
12	4.53937
13	4.81456
14	4.73314
15	5.37523
16	4.69305
17	4.77252
18	4.69235
19	4.80061
20	4.7014
21	4.94708
22	4.82892
23	4.64056
24	4.61854
25	5.07818

Var. of Neighbors	Stable State Var. = 4.53029
Neighbors of Stable State (26-50)	Variance
26	5.09535
27	4.81611
28	4.9284
29	4.82502
30	4.75446
31	4.76976
32	4.71299
33	5.41648
34	4.84583
35	5.39094
36	4.60564
37	4.65822
38	4.87447
39	4.66534
40	4.60035
41	4.54065
42	4.73684
43	4.55279
44	4.56739
45	5.1763
46	4.67119
47	4.97825
48	4.88673
49	5.03052
50	4.65258

Appendix

Listing 1: Code:2

```
 ///////////////////////////////////////////////////////////////////////////////////
// This is starter code for Homework 6.
///////////////////////////////////////////////////////////////////////////////////

// This function is found below.
void GetData ();

// "ticker" is a global variable.
char **ticker;

// The covariance matrix is a global variable.
double **V;
//Global Variables
double** Weights;
//double** WeightsCopy;
double* Best;
double E, U, U1, U2;
int BuyPenny, SellPenny;

//Declared functions
void AllocateMemory();
void DeleteMemory();
void InitializePortfolio();
void NeighborProposal(double**);
void Metropolis();
double Energy(double**);

#include <math.h>     // various math functions, such as exp()
#include <stdio.h>    // various "input/output" functions
#include <time.h>     // functions for timing computations
#include "Functions.h" //functions given
#include <iostream>

int main () {

   int i, seed;

   // Read in and display the tickers and covariance matrix.
   GetData ();

   // Show the 50 tickers.
   for (i = 1; i <= 50; i++) {
      printf (ticker[i]);
   }
   Pause ();

   // Seed the RNG.
   seed = GetInteger ("RNG␣Seed?...␣");
   U = MTUniform (seed);

   ////////////////////////////////////////////////////////////////////////////////
   // Your Metropolis algorithm goes here...

   //Function calls go here.
   AllocateMemory();

   InitializePortfolio();

   Metropolis();

   DeleteMemory();

   // Pause so the execution window does not close.
   Exit ();

}



////////////////////////////////////////////////////////////////////////////////
// Allocate space for and read in covariance matrix and stock tickers.
////////////////////////////////////////////////////////////////////////////////
void GetData () {

   int i, j;
   double V0;
   char input[100];
   FILE *fp;

   // Allocate array space.
   V = Array (50, 50);

   // Allocate space to hold ticker names; "ticker" is a global variable.
   ticker = (char **) calloc (50+1, sizeof (char *));
   for (i = 0; i <= 50; i++) {
      ticker[i] = (char *) calloc (10, sizeof (char));
   }

   // Read in the data.
   fp = fopen ("V.txt", "r");
   for (i = 1; i <= 50; i++) {

      // Read in stock i's covariance data.
```

```
        // Name of the stock ticker.
        fgets (ticker[i], 9, fp);

        // The 50 covariances for stock "i".
        for (j = 1; j <= 50; j++) {

            // Read in V[i][j].
            fgets (input, 99, fp);
            sscanf (input, "%lf", &V0);

            // Put data into the V array.
            V[i][j] = V0;

        }

    }
    fclose (fp);

    return;

}
// Create Memory for the weights of Portfolio
// and the best Portfolio
void AllocateMemory(){
  Weights = new double*[51];
  Best = new double[51];

  for(int i = 1; i<51; ++i){
    Weights[i] = new double[1];
  }

  Weights[0] = new double[2];
  Weights[0][0] = 50; Weights[0][1] = 1;
}
// Delete the memory allocated for calculations
void DeleteMemory(){
  delete[] Best;
  for(int i = 0; i<51; ++i){
    delete[] Weights[i];
  }
  delete[] Weights;
}
//Random generate a initial Portfolio
void InitializePortfolio(){
  double allocation, sum, total;
  int a;
  a = 1; // a counter variable
  sum = 0; // total current allocation is 0
```

```
    while(1){ // while on
      // pick a number between - 10000 and 10000
      allocation = int(10000 * (-1 +  2 * U));
      // put into dollars
      allocation = allocation/100.0;
      // if at 50th index;
      if(a==50){
        // buy long or short to bring you to $100
        // Weights[a][1] + sum = 100 implies =>
        Weights[a][1]= 100 - sum;
        break;
      }
      // if we arent at the end
      // update total current allocation
      sum += allocation;
      // set allocation
      Weights[a][1]=allocation;
      // generate next uniform
      U = MTUniform(0);
      // increment a
      ++a;
    }
  }

  void Metropolis () {
    //create variables
     double T, DeltaE , p, E_min;
     int k, n, AcceptTransition , NextReport;

     // Get the temperature.
     T = GetDouble ("What␣is␣the␣temperature?...␣");
     E = Energy(Weights);
     E_min = 100000000000;
     for(int j = 0; j<50; ++j){
       std::cout<<"Weight␣for␣"<<ticker[j+1]<<":␣"<<Weights[j+1][1]<<"\n";
     }
     Pause();

     // Time the computations.
     Time ();

     // Simulate the Markov chain 100 million steps. The portfolio  will
     //   always hold the current value of the Markov chain X0, X1, X2,... .
     for (n = 1; n <= 10000000; n++) {

        // Get a proposed random change.
        NeighborProposal (Weights);

        // Compute the change in energy associated with reversing that portion
        //    of the portfolio.

        DeltaE =  Energy(Weights) - E;

        // See if the proposed transition is accepted.
        AcceptTransition = 0;

        if (DeltaE <= 0) { // if decrease in energy
           AcceptTransition = 1;
        }
        else if (T > 0) { // calculate mu(y)/mu(x)
           p = exp(-DeltaE / T);
           U = MTUniform(0);
           if (U <= p) {
              AcceptTransition = 1;
           }
        }

        // Effect the transition. If the new energy is a best-so-far, record the
        //     portfolio (in best[*]) and the new minimal energy (Emin).
        if (AcceptTransition==1) {
           // Update the variance of the current portfolio (the portfolio's "energy").
           E += DeltaE;
           // Record data for the best portfolio, if appropriate.
           if (E < E_min) {
              E_min = E;
              for (k = 0; k < 50; k++) {
                 Best[k] = Weights[k+1][1];
              }
           }
        } // End of "if" statement.
        else{
          // if we didnt accept the transition,
          // reverse changes done to the pointer
          Weights[BuyPenny][1] = Weights[BuyPenny][1]-0.01;
          Weights[SellPenny][1] = Weights[SellPenny][1]+0.01;
        }

        if(n == 100000){
          printf ("Should be done in %.1f seconds.\n", (100)*Time());

        }
        // Show progress to the screen.
        if (n % 10000 == 0) {
           printf (". ");
        }

    } // This ends the Markov chain simulation loop.
```

```
   // Report the best portfolio found throughout the Markov chain.
   E = E_min;

   printf ("\n\n");
   for(int j = 0; j<50; ++j){
     std::cout<<"Weight for "<<ticker[j+1]<<": "<<Best[j]<<"\n";
   }
   std::cout<<"The minimum variance is "<<E<<".\n";
   printf ("\n\n");
   printf ("100 hundred million Markov chain steps completed in %.1f seconds.\n", Time ())
   //printf ("View the solution with ShowRoutes.tex.\n");
}

////////////////////////////////////////////////////////////////////////////////
// Determine the proposed transition. ///////////////////////////////////////////
void NeighborProposal (double** CurrentWeights) {
   // generate a uniform
   U1 = MTUniform(0);
   //pick an index to buy a penny
   BuyPenny = int(U1*50)+1;
   while(1){ //while on
     // generate another uniform
     U2 = MTUniform(0);
     // pick that index to sell a penny
     SellPenny = int(U2*50)+1;
     // if it is different, then select it.
     if(BuyPenny != SellPenny){
       break;
     }
   }
   // Increment weights by a penny
   Weights[BuyPenny][1] = CurrentWeights[BuyPenny][1] + 0.01;
   Weights[SellPenny][1] = CurrentWeights[SellPenny][1] - 0.01;
 }

////////////////////////////////////////////////////////////////////////////////
// Determine Energy of New Portfolio
// function is a variation of the given variance function.
double Energy(double** Portfolio){
   double energy;
   double** IntermediateProduct;
   double** Product;
   double** TransposePortfolio;
   TransposePortfolio = Transpose(Portfolio);
   IntermediateProduct = Multiply(V, Portfolio);
   Product = Multiply(TransposePortfolio, IntermediateProduct);
   energy = Product[1][1];
   delete[] IntermediateProduct;
   delete[] Product;
```

```
    delete[] TransposePortfolio;
    return energy;
}
```

Appendix

Listing 2: Code:3

```
void InitializePortfolio(){
  // create needed variables
  double allocation, sum, total;
  int a;
  a = 1; // counter
  sum = 0; // total allocation
  total = 10000; // total dollars available * 100
  while(1){ // while on
    // choose a random amount of dollars
    allocation = int(total * U);
    // subtract allocation from total available money
    total -= allocation;
    // put into dollars
    allocation = allocation/100.0;
    // if at last stock, i will put a penny in it.
    // due to the interger method on the allocation
    if(a==50){
      Weights[a][1]= 100 - sum;
      break;
    }
    sum += allocation; // keep running total of allocation
    // set weight
    Weights[a][1]=allocation;
    //Generate next uniform
    U = MTUniform(0);
    // increment a
    ++a;
  }
}

////////////////////////////////////////////////////////////////////////////////
// Determine Energy of New Portfolio
// function is a variation of the given variance function.
double Energy(double** Portfolio){
   double energy;
   // check if any stocks are negative
   for(int i = 1; i<51; ++i){
      if(Portfolio[i][1]<-0.005){
        // if negative then set energy to something unreasonable
        energy = 100000;
        return energy;
      }
   }
   // otherwise same function
   double** IntermediateProduct;
```

```
    double** Product;
    double** TransposePortfolio;
    TransposePortfolio = Transpose(Portfolio);
    IntermediateProduct = Multiply(V, Portfolio);
    Product = Multiply(TransposePortfolio, IntermediateProduct);
    energy = Product[1][1];
    delete[] IntermediateProduct;
    delete[] Product;
    delete[] TransposePortfolio;
    return energy;
}
```

Appendix

Listing 3: Code:4

```
///////////////////////////////////////////////////////////////////////////////
// This is starter code for Homework 6.
///////////////////////////////////////////////////////////////////////////////

// This function is found below.
void GetData ();

// "ticker" is a global variable.
char **ticker;

// The covariance matrix is a global variable.
double **V;
//Global Variables
double** Weights;
double** WeightsCopy;
double* Best;
double E, U, U1, U2;
int BuyPenny, SellPenny;

//Declared functions
void AllocateMemory();
void DeleteMemory();
void InitializePortfolio();
void NeighborProposal();
void Metropolis();
double Energy(double**);

#include <math.h>     // various math functions, such as exp()
#include <stdio.h>    // various "input/output" functions
#include <time.h>     // functions for timing computations
#include "Functions.h" //functions given
#include <iostream>

int main () {

   int i, seed;

   // Read in and display the tickers and covariance matrix.
   GetData ();

   // Show the 50 tickers.
   for (i = 1; i <= 50; i++) {
      printf (ticker[i]);
   }
   Pause ();
   // Seed the RNG.
   seed = GetInteger ("RNG␣Seed?...␣");
   U = MTUniform (seed);

   ////////////////////////////////////////////////////////////////////////////////
   // Your Metropolis algorithm goes here...

   //Function calls go here.
   AllocateMemory();

   InitializePortfolio();

   Metropolis();

   DeleteMemory();

   // Report the best-found portfolio and its variance here.

   // Pause so the execution window does not close.
   Exit ();

}



////////////////////////////////////////////////////////////////////////////////
// Allocate space for and read in covariance matrix and stock tickers.
////////////////////////////////////////////////////////////////////////////////
void GetData () {

   int i, j;
   double V0;
   char input[100];
   FILE *fp;

   // Allocate array space.
   V = Array (50, 50);

   // Allocate space to hold ticker names; "ticker" is a global variable.
   ticker = (char **) calloc (50+1, sizeof (char *));
   for (i = 0; i <= 50; i++) {
      ticker[i] = (char *) calloc (10, sizeof (char));
   }

   // Read in the data.
   fp = fopen ("V.txt", "r");
   for (i = 1; i <= 50; i++) {
        // Read in stock i's covariance data.

        // Name of the stock ticker.
        fgets (ticker[i], 9, fp);

        // The 50 covariances for stock "i".
        for (j = 1; j <= 50; j++) {

            // Read in V[i][j].
            fgets (input, 99, fp);
            sscanf (input, "%lf", &V0);

            // Put data into the V array.
            V[i][j] = V0;

        }

    }
    fclose (fp);

    return;

}
// create memory
void AllocateMemory(){
  // create a copy of weights for step reversion
  Weights = new double*[51];
  WeightsCopy = new double*[51];
  Best = new double[51];

  for(int i = 1; i<51; ++i){
    Weights[i] = new double[1];
    WeightsCopy[i] = new double[1];
  }

  Weights[0] = new double[2];
  Weights[0][0] = 50; Weights[0][1] = 1;

  WeightsCopy[0] = new double[2];
  WeightsCopy[0][0] = 50; WeightsCopy[0][1] = 1;
}
// delete memory
void DeleteMemory(){
  delete[] Best;
  for(int i = 0; i<51; ++i){
    delete[] Weights[i];
    delete[] WeightsCopy[i];
  }
  delete[] Weights;
  delete[] WeightsCopy;
}
// initaialize random portfolio
void InitializePortfolio(){
  // create variables
  int zeroes, zeroesLeft, a;
  double leftspace, p;
  // choose a random number of zero weight stocks to have in portfolio
  zeroes = int(U * 41);
  // dummy variable to know when to stop allocating money.
  zeroesLeft = zeroes;
  std::cout<<"Zeroes: "<<zeroes<<"\t Non-Zero Indexes: "<<50-zeroes<<"\n";
  a = 0; // counter
  while(a<50){ // while not at the end
    // calculate how much space is left in array
    leftspace = 50 - a;
    // the remaining zeroes that havent been allocated to a position
    // divided by the remaining space in the array will give a probability
    // for choosing a zero weight to allocate
    p = zeroesLeft / leftspace;
    // generate uniform
    U1 = MTUniform(0);
    // if less than that probability
    if(U1<p){
      // set weight to 0
      Weights[a+1][1] = 0;
      // take away zero to allocate somewhere
      --zeroesLeft;
      // increment a
      ++a;
    }
    else{
      // allocate an equal weight to the position
      Weights[a+1][1] = 100.0 / (50 - zeroes);
      // increment a
      ++a;
    }
  }
}

void Metropolis () {
   // create variables
   double T, DeltaE, p, E_min;
   int k, n, AcceptTransition, NextReport;

   // Get the temperature.
   T = GetDouble ("What is the temperature?... ");
   E = Energy(Weights);
   E_min = 100000000000;
```

```
for(int j = 0; j<50; ++j){
  std::cout<<"Weight␣for␣"<<ticker[j+1]<<":␣"<<Weights[j+1][1]<<"\n";
}
Pause();

// Time the computations.
Time ();

// Simulate the Markov chain 100 million steps. The portfolio weights will
//   always hold the current value of the Markov chain X0, X1, X2,... .
for (n = 1; n <= 10000000; n++) { // sim loop
   // copy weights to revert back if needed
   for(int i = 1; i < 51; ++i){
     WeightsCopy[i][1] = Weights[i][1];
   }
   // Get a proposed random change.
   NeighborProposal ();
   // Compute the change in energy associated with reversing that portion
   //   of the portfolio.
   DeltaE =  Energy(Weights) - E;
   // See if the proposed transition is accepted.
   AcceptTransition = 0;

   if (DeltaE <= 0) { // if decrease in energy
      AcceptTransition = 1;
   }
   else if (T > 0) { // calculate mu(y)/mu(x)
      p = exp(-DeltaE / T);
      U = MTUniform(0);
      if (U <= p) {
         AcceptTransition = 1;
      }
   }

   // Effect the transition. If the new energy is a best-so-far, record the
   //     portfolio (in best[*]) and the new minimal energy (Emin).
   if (AcceptTransition==1) {
      // Update the variance of the current portfolio (the portfolio's "energy").
      E += DeltaE;
      // Record data for the best portfolio, if appropriate.
      if (E < E_min) {
         E_min = E;
         for (k = 0; k < 50; k++) {
            Best[k] = Weights[k+1][1];
         }
      }
   } // End of "if" statement.
   else{
     // else revert back to old
```

```
        for(int i = 1; i < 51; ++i){
          Weights[i][1] = WeightsCopy[i][1];
        }
      }

      if(n == 1000000){
        printf (”Should␣be␣done␣in␣%.1f␣seconds.\n”, (10)*Time());
      }
      // Show progress to the screen.
      if (n % 100000 == 0) {
         //Pause();
         printf (”.␣”);
      }
   } // This ends the Markov chain simulation loop.

   // Report the best route found throughout the Markov chain.
   E = E_min;

   // Finish up; report best-found route length to the screen.
   printf (”\n\n”);
   for(int j = 0; j<50; ++j){
     if(Best[j]!=0){
       std::cout<<”Weight␣for␣”<<ticker[j+1]<<”:␣”<<Best[j]<<”\n”;
     }
   }
   int zeroes = 0;
   // counting zeroes
   for(int i = 1; i<51; ++i){
     if(Best[i]==0){
       ++zeroes;
     }
   }
   std::cout<<”Zeroes:␣”<<zeroes<<”\t␣Non-Zero␣Indexes:␣”<<50-zeroes<<”\n”;
   std::cout<<”The␣minimum␣variance␣is␣”<<E<<”.\n”;
   printf (”\n\n”);
   printf (”100␣hundred␣million␣Markov␣chain␣steps␣completed␣in␣%.1f␣seconds.\n”, Time ())
   //printf (”View the solution with ShowRoutes.tex.\n”);
}

/////////////////////////////////////////////////////////////////////////////////
// Determine the proposed transition. ////////////////////////////////////////////
void NeighborProposal () {
  // create variables
   int zeroes = 0;
   // count zeroes
   for(int i = 1; i<51; ++i){
     if(Weights[i][1]==0){
       ++zeroes;
     }
```

```
}
// if at boundary of 10 stocks only
if(zeroes==40){
  // generate uniform
  U1 = MTUniform(0);
  // pick index
  BuyPenny = int(U1*50)+1;
  // if you pick an index that is allocated
  // force another index to be picked
  while(Weights[BuyPenny][1]!=0){
    U1 = MTUniform(0);
    BuyPenny = int(U1*50)+1;
  }
  // give that zero weight index some allocation
  Weights[BuyPenny][1] = 1;
  // one less zero weight index
  --zeroes;
  // reallocate portfolio
  for(int i = 1; i<51; ++i){
    if(Weights[i][1]!=0){
      Weights[i][1] = 100.0/(50-zeroes);
    }
  }
  //Pause();
}
// else at other boundary of every stock having allocation
else if(zeroes==0){
  // generate uniform
  U1 = MTUniform(0);
  // pick an index
  BuyPenny = int(U1*50)+1;
  // take away its allocation
  Weights[BuyPenny][1] = 0;
  // one more zero weight index
  ++zeroes;
  // reallocate portfolio
  for(int i = 1; i<51; ++i){
    if(Weights[i][1]!=0){
      Weights[i][1] = 100.0/(50-zeroes);
    }
  }
}
else{
  // else in middle, pick randomly to add zero Indexes
  // or to take away a zero index
  U2 = MTUniform(0);
  // flip coin
  if(U2<0.5){
    // pick index to assign weight
```

```
        U1 = MTUniform(0);
        BuyPenny = int(U1*50)+1;
        while(Weights[BuyPenny][1]!=0){
          U1 = MTUniform(0);
          BuyPenny = int(U1*50)+1;
        }
        Weights[BuyPenny][1] = 1;
        --zeroes;
        // reallocate portfolio
        for(int i = 1; i<51; ++i){
          if(Weights[i][1]!=0){
            Weights[i][1] = 100.0/(50-zeroes);
          }
        }
      }
      else{
        // pick index to take allocation
        U1 = MTUniform(0);
        BuyPenny = int(U1*50)+1;
        while(Weights[BuyPenny][1]==0){
          U1 = MTUniform(0);
          BuyPenny = int(U1*50)+1;
        }
        Weights[BuyPenny][1] = 0;
        ++zeroes;
        //reallocate portfolio
        for(int i = 1; i<51; ++i){
          if(Weights[i][1]!=0){
            Weights[i][1] = 100.0/(50-zeroes);
          }
        }
      }
    }
 }


////////////////////////////////////////////////////////////////////////////////
// Determine Energy of New Portfolio
// variation of given variance function
double Energy(double** Portfolio){
   double** IntermediateProduct;
   double** Product;
   double** TransposePortfolio;
   TransposePortfolio = Transpose(Portfolio);
   double energy;
   IntermediateProduct = Multiply(V, Portfolio);
   Product = Multiply(TransposePortfolio, IntermediateProduct);
   energy = Product[1][1];
   delete[] IntermediateProduct;
   delete[] Product;
   delete[] TransposePortfolio;
   return energy;
}
```

Appendix

Listing 4: Code:5

```
// create function to output neighbors energy
 void NeighborsEnergy(){
   // create variable
   double OGenergy;
   int zeroesCalculation;
   int zeroes = 0;
   int done = 0;
   // store original portfolios energy
   OGenergy = Energy(Weights);
   std::cout<<"The Energy of the state is: "<<OGenergy<<"\n";

   // copy to revert back
   for(int i = 1; i < 51; ++i){
     WeightsCopy[i][1] = Weights[i][1];
   }
   // count zeroes
   for(int i = 1; i<51; ++i){
     if(Weights[i][1]==0){
       ++zeroes;
     }
   }
   // store dummy variable for calculations
   zeroesCalculation = zeroes;
   // go through each index
   for(int j = 1; j<51; ++j){
     // if allocated, give it no allocation
     if(Weights[j][1]!=0){
       Weights[j][1] = 0;
       ++zeroesCalculation;
       // reallocate portfolio to get the neighbor
       for(int i = 1; i<51; ++i){
         if(Weights[i][1]!=0){
           Weights[i][1] = 100.0/(50-zeroesCalculation);
         }
       }
       // output neighboring portfolios energy
       std::cout<<"Neighoring State: "<<j<<" Energy: "<<Energy(Weights)<<"\n";
       // reset weights of portfolio
       for(int i = 1; i < 51; ++i){
         Weights[i][1] = WeightsCopy[i][1];
       }
       // reset zeroes
       zeroesCalculation = zeroes;
     }
```

```
    else{
      // else it doesnt have allocation,
      // so give it allocation
      Weights[j][1] = 1;
      --zeroesCalculation;
      // reallocate portfolio to get the neighbor
      for(int i = 1; i<51; ++i){
        if(Weights[i][1]!=0){
          Weights[i][1] = 100.0/(50-zeroesCalculation);
        }
      }
      // output neighboring portfolios energy
      std::cout<<"Neighoring␣State:␣"<<j<<"␣Energy:␣"<<Energy(Weights)<<"\n";
      // reset weights of portfolio
      for(int i = 1; i < 51; ++i){
        Weights[i][1] = WeightsCopy[i][1];
      }
      // reset zeroes
      zeroesCalculation = zeroes;
    }
  }
}
```

Our group separated the work relatively uniformly over the interval of time required to complete the assignment. We met on a regular basis and worked through each problem at a time, always listening to criticism and different points of view. We all coded together, and if something was unclear to someone when writing the code, we always explained precisely what was happening line by line. Our group has a great interest in these projects and subject matter; therefore, we have had no issues with scheduling and collaborating as a team. We genuinely enjoy working on these projects together.