COMPUTATIONAL ECONOMICS

Portfolio Model in MATLAB

David A. Kendrick
Ruben P. Mercado
Hans M. Amman

1 Introduction

The classic portfolio optimization problem, originally proposed by Markowitz (1952), was to consider both the mean and the variance of a portfolio by maximizing the mean while minimizing the variance. This was formulated as a quadratic programming problem to maximize a weighted sum of the mean and the negative of the variance. Thus one could consider the trade-off between stocks with high means and greater risk with their higher variances and stocks with low means and low risk with lower variances. Moreover, one might think in terms of building a diversified portfolio that contains stocks that tend to move in opposite directions as represented by negative covariance elements.

Our goal in this chapter is to use **MATLAB** to solve the optimal portfolio problem. First, we solve the problem using a simple Monte Carlo optimization search program. This is useful to provide a simple introduction to the **MATLAB** programming language and at the same time to learn a little about a random search procedure for optimization.

Then, we move on to solve the portfolio optimization problem using a **MATLAB** gradient optimization function. However, this code makes use of the Optimization Toolbox, which may not be available to all users of **MATLAB**, so we provide a GAMS version of the same problem in Apendix E. Some readers may find the GAMS version somewhat easier to understand and can use it as an entry ramp to the **MATLAB** gradient optimization program.[1]

2 The Mathematics

Consider a vector whose elements are the fractions of the portfolio that are invested in each of the equities, that is,

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{1}$$

where x_i is the fraction of the portfolio invested in equity i for an example portfolio with three equities. There is also a vector μ that contains the mean return on each of the equities:

[1] Also some readers may want to solve models of higher dimension than those used in this chapter with modified versions of both the **MATLAB** code and the GAMS code in order to compare the computational speeds of the two software systems.

$$\mu = \begin{bmatrix} \mu_1 \\ \mu_2 \\ \mu_3 \end{bmatrix} = \begin{bmatrix} 8 \\ 12 \\ 15 \end{bmatrix} \tag{2}$$

where

$$\mu_i = \text{the mean return on equity } i$$

Note in this example that the second and third equities have the highest mean returns of 12 and 15, respectively. These data for the means in equation (2) and the covariances shown below are for illustrative purposes and do not represent the return on particular equities or groups of equities.

We can then use the inner product of these two vectors, that is,

$$\mu' x = \begin{bmatrix} \mu_1 & \mu_2 & \mu_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{3}$$

to obtain the mean return for the portfolio. The variance for the portfolio is given in the covariance matrix Σ, that is,

$$\Sigma = \begin{bmatrix} \sigma_{11} & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_{22} & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_{33} \end{bmatrix} = \begin{bmatrix} 6 & -5 & 4 \\ -5 & 17 & -11 \\ 4 & -11 & 24 \end{bmatrix} \tag{4}$$

where

$$\sigma_{ij} = \text{the covariance of the returns on equities } i \text{ and } j$$

Note that here the second and third equities, which have the highest mean returns, also have the highest variances of 17 and 24, respectively. Also note that the off-diagonal elements in the covariance matrix have different signs. Thus, for example, when the return on the first equity falls, the return on the second tends to rise since the covariance is -5. Thus holding these two equities in the same portfolio provides a cushion when the return on the first equity declines and the return on the equity rises.

The variance of the portfolio can then be written as

$$x'\Sigma x = \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} \sigma_{11} & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_{22} & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \tag{5}$$

The Markowitz model considers both the mean and the variance of a portfolio by maximizing the mean while minimizing the variance. Using the components of the mean and variance of the portfolio from equations (3) and (5), one can write the criterion function for the model as to maximize J in

$$J = \mu' x - \frac{1}{2}\beta x' \Sigma x \tag{6}$$

where

$$\begin{aligned} J &= \text{criterion value} \\ \beta &= \text{subjective weight on the variance of the return on the portfolio} \end{aligned}$$

The parameter β provides the subjective weight on the variance. Thus an individual with a high β is risk averse and will choose a portfolio with equities that have relatively small variances. The $\frac{1}{2}$ in this expression is commonly used in quadratic criterion functions, but it plays no essential role.

The constraint for this model simply requires that the fractions invested in each of the equities add up to one, that is,

$$\sum_{i \in I} x_i = 1 \tag{7}$$

where

$$I = \text{the set of equities}$$

There is also a constraint that requires that the factions be nonnegative, that is,

$$x_i \geqslant 0 \quad i \in I \tag{8}$$

So, in summary, the model is to find the values of x_i that maximize J in equation (6), subject to the constraints in equations (7) and (8).

The optimal portfolio model can also be posed in a related way that seeks to find the fractional equity holdings that minimize the weighted risk subject to a constraint that the mean return on the portfolio be above a specified level. The criterion function for this model is

$$J = \frac{1}{2}\beta y' \Sigma \tag{9}$$

where

$$y = \text{ vector of fractions of portfolio invested in each equity}$$

subject to

$$\mu' y \geqslant \theta \tag{10}$$

where

$$\theta = \text{ desired minimum mean return on portfolio} \tag{11}$$

$$\sum_{i \in I} y_i = 1 \tag{12}$$

$$y_i \geqslant 0 \quad i \in I \tag{13}$$

In summary, this second version of the model is designed to find the values of y_i that minimize J in equation (9), subject to the constraints in equations (10) through (13). The key parameter in this formulation is θ, the desired minimum mean return for the portfolio. As this parameter is increased the optimal portfolio includes more of the risky equities.

This completes the statement of the mathematics of the two versions of the models. Next we turn to the computational statement of the models in **MATLAB**.

3 A Simple Monte Carlo Optimization Procedure in MATLAB

In order to provide a good opportunity to learn the basics of both the **MATLAB** software and a Monte Carlo optimization search procedure we have chosen a simple application to solve the first formulation of the Markowitz model, which is based on some programs developed by our students Paul Maksymonko, Kevin Kline, and Carter Hemphill.[2] The optimization procedure includes the generation of a population of eight candidate portfolios in the first period. The portfolio that performs best is then selected and stored. Then the next-period

[2]They developed some applications to be used as an introduction to an optimization method known as genetic algorithms, which is particularly useful when dealing with nonconvex problems. We do not deal with that method here, although the application we present resembles that method. The approach used here is more like an evolutionary algorithm (EA) in that it uses real numbers rather than the strings of bits that are used in many genetic algorithms (GA). Later in the book we provide an introduction to genetic algorithms in a chapter on that subject.

portfolios are generated as random variations around that portfolio. This process is repeated 100 times.

The basic structure of the program is a set of two for loops. The outside loop is across time periods (or runs) and the inside loop is over candidates. These loops look something like the following:

```
nruns = 100 ; popsize = 8;
for k = 1:nruns;
    for i = 1:popsize;
        .....
          end
          .....
end
```

Note that the indentation in the foregoing code makes it easy to see the beginning and ending of each of the for loops. The indentation is not necessary for the **MATLAB** compiler but can (and should) be used to make the code easier to read.

The first step in the code is to initialize the number of time periods (or runs) and the number of candidates in each time period with the **MATLAB** statements

```
nruns = 100; popsize = 8;
```


This is followed by the main for loop in the program, which is over the time periods (or runs). The structure of the for loop is

```
for k =1:nruns;
    main body of the program
end
```

Thus the time index in this model is k and it runs from one to the number of runs. Note further that each for loop extends until the matching end statement is encountered, so it is useful when reading **MATLAB** code of this type to examine the structure of the code by looking for matching for and end statements. This is shown below in a pseudocode outline of the structure of the program. The code is called pseudo because it cannot be run on a computer as it is, but rather is intended to outline the basic structure of the program.

```
nruns = 100 ; popsize = 8;
initialize portfolio weights
for k = 1:nruns;
    generate returns, variance costs and criterion values
    select best portfolio
```

```
    for i = 1:popsize;
                    generate new random portfolio weights (percentages) for each candidate
    end
end

print and graph the sequence of best candidates
```

After the number of runs and the population size are set the initialization section of the code is used to set the initial portfolio weights for each candidate.

Then the k loop for the number of runs begins with the computation for that time period of the returns and the variance costs for each of the portfolios. These values are then used to calculate a vector that gives the criterion value obtained for each of the eight candidates. This criterion vector is then examined to find the index of the candidate with the highest criterion value. This best portfolio is then used in the second, nested, i loop for the candidates as the basis for the generation of the portfolio holding of the eight candidates in the next period. After the time period (or run) loop is repeated 100 times the sequence of best portfolios in each period is printed and plotted.

With this overview of the program in mind we consider next each of the sections of the code.

4 Initialization of Counters, Parameters, and Weights

The initialization section of the code contains the initialization of the counters for the number of runs (`nruns`), the population size (popsize), and the parameters of the portfolio model [the risk aversion coefficient (`beta`), the vector of mean returns (`mu`), and the covariance matrix (`sigma`)]:

```
nruns = 100; popsize = 8;
beta = 2;
mu = [8 12 15]';

sigma = [6 -5 4
-5 17 -11
4 -11 24];
```

There is also a constant

```
const = 0.1;
```

that will be used later to determine the degree of random variation around the weights of the best candidate of a time period to generate the candidates of the next period. Finally, we create the vector of initial portfolios for the first time period with the statement

```
pwm = (1/3) * ones(3,popsize);
```

Thus pwm stands for the portfolio weight matrix. The function ones() generates a matrix of ones with three rows and a number of columns equal to the population size, so pwm contains eight column vectors with one portfolio each, all with weights set equal to $\frac{1}{3}$. The initial `pwm` looks like

$$pwm = \begin{bmatrix} .33 & .33 & .33 & .33 & .33 & .33 & .33 & .33 \\ .33 & .33 & .33 & .33 & .33 & .33 & .33 & .33 \\ .33 & .33 & .33 & .33 & .33 & .33 & .33 & .33 \end{bmatrix}$$

Note here that in **MATLAB** it is not necessary to first *declare* a variable and then *define* it. Declarations are used in many program languages to determine the type of a variable, namely whether it is an integer or a floating point number and whether it is a scalar or a multidimensional array. In addition, the declaration is used to set aside enough space in memory to store the elements of the variable before the numerical values of each element are defined in a separate statement in the language. Thus, in the statement

```
mu = [8 12 15]';
```

that is used above, the variable `mu` is both declared and defined by its context to be a column vector with three elements. The vector is input as a row vector but the transpose (′) mark is used to convert it to a column vector.

The next step is the generation of the returns, variance cost, and criterion value for each portfolio.

5 Generation of Returns, Variance Costs, and Criterion Values

The returns for every candidate are generated with the statement

```
pret = pwm' * mu;
```

where `pret` is an eight-element vector that contains the portfolio return for each of the eight candidates. The original `pwm` matrix is 3×8 as we saw earlier; therefore, its transpose, which is used in the foregoing statement is 8×3. This matrix in turn is multiplied by the three-element column vector mu to yield the eight-element column vector pret.

Generating the variance costs for every candidate requires the use of a short loop

```
for j = 1:popsize;
    pvar(j) = 0.5 * beta * pwm(:,j)' * sigma * pwm(:,j);
end
```

The notation `(:,j)` in the matrix pwm refers to all the elements of the *j*th column of the matrix. Remember that each column in pwm corresponds to one portfolio. Thus by the time the code has passed through this loop eight times the variance costs for all the candidates are neatly stored in the pvar vector, which has eight elements (one for each candidate).

The criterion values for each candidate are just the differences between the portfolios returns and the variance costs and are computed with the statement

```
pcrit = pret - pvar';
```

The vectors pret and pvar each have eight elements. Since the vector pvar is a row vector it has to be transposed in the foregoing expression. Thus pcrit is an eight-element column vector that contains the criterion value for each of the portfolios. Thus this vector can be used to find the best candidate. Of course in the first pass through this part of the code all the portfolios are the same, so all the criterion values are the same as well..

6 Selection of the Best Portfolio

The next step is to find the portfolio that has the highest rate of return. This is done with the statement

```
[top topi] = max(pcrit);
```

which uses the **MATLAB** function max to place in the scalar top the largest element in the vector pcrit and the corresponding index in the scalar topi. If there is more than one maximum, this function chooses only one.

The index is then used to put the set of portfolio weights used by this candidate into the vector wnew with the statement

```
wnew = pwm(:,topi);
```

Recall that the matrix `pwm` has three rows (one for each asset class) and eight columns (one for each candidate) so the effect of the statement above is to put the three elements from the `topi` column of the matrix into the vector `wnew`. The portfolio weights for the best candidate and the criterion value in each time period k are then stored in the matrices wbest and pcritvec using the statements

```
wbest(:,k) = wnew;
pcritvec(:,k) = top;
```

These arrays can then be used at the bottom of the code to plot the best portfolio in each run and the corresponding criterion value.

7 Random Generation of New Portfolios

The candidates for the next period are created as random variations around the portfolio weights of the best candidate from the previous period. The weights from the best candidate have been stored in the vector wnew and that vector is used in the for loop below to create eight new candidates:


```
for i = 1:popsize-1;
     w1 = wnew(\ref{eq8-1}) + rand * const;
           w2 = wnew(2) + rand * const;
           w3 = wnew(3) + rand * const;
           temp = w1 + w2 + w3;
           w1 = w1/temp;
           w2 = w2/temp;
           w3 = w3/temp;
           pwnew(:,i) = [w1 w2 w3]';
end
```

The **MATLAB** random number generator, `rand`, for uniform distributions between zero and one is used here and is multiplied by a constant. This has the effect of adding a given amount to the portfolio weight for each equity. The weights are then normalized so that they add up to one. The last statement in the loop above, that is,

```
pwnew(:,j) = [w1;w2;w3];
```

simply stores the weight vector for the jth candidate in the jth column of the new portfolio weight matrix, pwnew. Thus by the time the loop has been completed the portfolio weights for the first seven candidates have been stored

in the pwnew matrix.

The next step is to put the best portfolio from the previous run in the last (eighth) column of the pwnew matrix using the statement

```
pwnew(:,popsize) = wnew ;
```

This has the effect of keeping the best solution from each run when generating the new portfolios to be used for the next run.

Then the statement

```
pwm = pwnew
```

is used to replace the previous-period matrix of portfolios by the newly generated matrix. Following this statement is the last end statement in the code. This is the end that corresponds to the for loop across time periods.

After the time period loop is completed the weights for the surviving candidate and the criterion values are printed with the simple statements

```
wnew
top
```

The absence of a semicolon at the end of these statements dictates that the result be printed. Finally, the following commands generate a graph displaying the values of the three assets percentage holdings for the best candidate in each time period:

```
xaxis = [1:1:nruns]';
plot(xaxis,wsurv(:,:));
xlabel('Runs');
ylabel('Weights');
legend('w1', 'w2', 'w3');
```

We have also commented out an additional statement that can be used to plot the criterion value for all runs:

```
plot(xaxis,pcritvec(:,:));
```

The entire code of the program is contained in Appendix A at the end of this chapter and is also available in the book web site under the name mcportfol.m. The instructions for running **MATLAB** are in Appendix B at the end of this book.

It is important to point out that every time you run the program, particularly when changing the number of runs or the population size, you should clean out the old commands and workspace to avoid displaying spurious results. To do so, go to Edit in the top **MATLAB** menu. Then select Clear Command Window and confirm with Yes that you want to do this. Then do the same for Clear Command History and for Clear Workspace. Alternatively, adding the sentence

```
clear all;
```

at the beginning of the program clears the workspace.

Figure 1 shows the sequence of weights of the best portfolios at each time period. The optimal portfolio weights for this experiment correspond to the last time period: $w1 = 0.24$, $w2 = 0.43$, and $w3 = 0.33$. Note that for this particular model and starting conditions that portfolios close to the optimum are found within only about ten runs.

This small random search optimization routine is simple to program and for our particular example problem is relatively effective in finding the optimal solution. More important, it serves our purpose of introducing the **MATLAB** software with a relatively uncomplicated code that performs nicely on this simple problem.

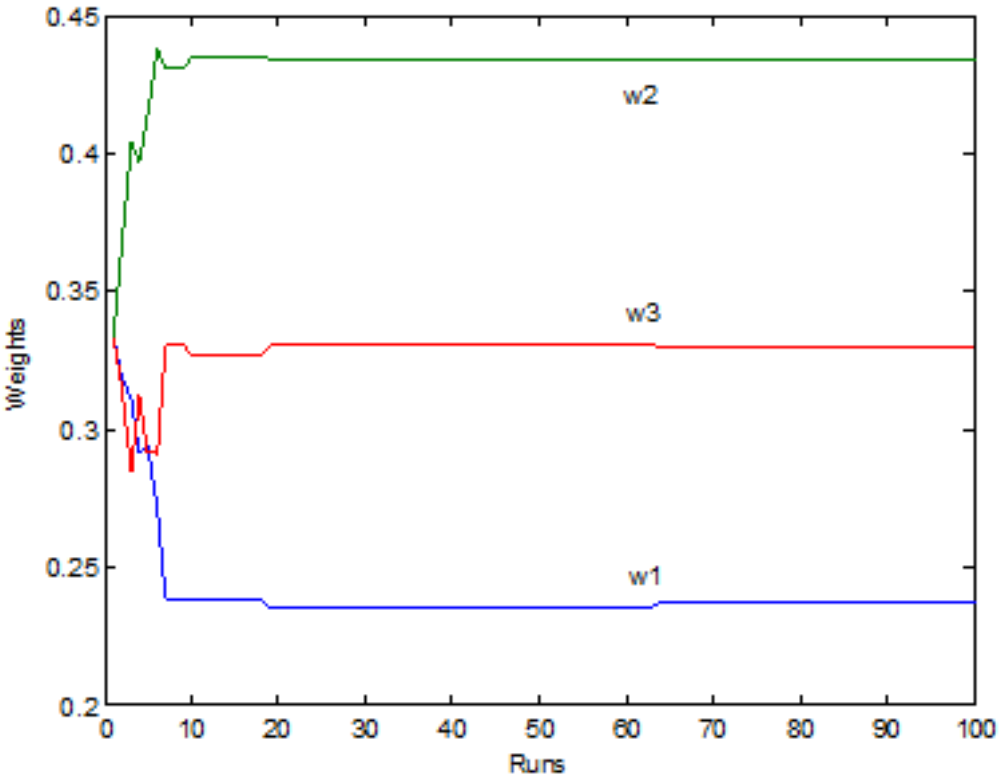
However, to see the shortcoming of this simple code you can try solving the case where beta is set equal to zero. In this case the solution is a boundary solution since the optimal portfolio is one in which the entire portfolio is placed in the one equity with the highest mean return. The previous simple code has a difficult time finding this solution, but the more complex gradient method approach discussed in the next section finds that optimal solution with relative ease.

8 The Markowitz Model Using a MATLAB Optimization Function

We turn now to the solution of both versions of the Markowitz model using a **MATLAB** function from the Optimization Toolbox, so before beginning to work with this code, be sure that the version of **MATLAB** that you are using includes the Optimization Toolbox.

The function to be used is fmincon, which is designed to find the minimum

Figure 1: Sequence of weights



of a function $f(x)$ with linear inequality and equality constraints and with nonlinear constraints. Thus our model can be solved with a nonlinear optimization solver (see Appendix D). A simplified version of this function call for a model that has only linear inequality constraints (this function call is used only for exposition and will not necessarily work in a **MATLAB** program) would be

```
[x,fval] = fmincon(@func,x0,A,b)
```

where

`x`	=	the vector of optimal values
`fval`	=	the value of the criterion function at the optimum
`fmincon`	=	the name of the function from the Optimization Toolbox
`func`	=	the name of the user supplied function that returns the criterion value for the function
`x0`	=	a vector of starting values to be used in the search for the optimal value of the function
`A`	=	the matrix for the linear inequalities `Ax <= b`
`b`	=	the vector for the linear inequalities `Ax <= b`

To use the fmincon function in this case you would have to supply a function

```
func
```

that would return the value of the criterion function. In addition, you should provide a vector `x0` of values that you think are close to the optimal value of the function. The vector is used by the Optimization Toolbox function as a starting point in the search for the optimal value of the function and also the parameters of the problem. You must also supply the `A` matrix and the `b` vector for the linear inequality

$$Ax <= b$$

that constrains the solution to the model.

A somewhat more complicated version of the call to fmincon would include, in addition to the linear inequalities, the linear equalities and the upper and lower bounds on the variables, and would be of the form

```
[x,fval]=fmincon(@func,x0,A,b,Aeq,beq,lb,ub);
```

where

`Aeq`	=	the matrix for the linear equalities `Aeq x = beq`
`beq`	=	the vector for the linear equalities `Aeq x = beq`
`lb`	=	lower bound on the variables `lb <= x`
`ub`	=	upper bound on the variables `x <= up`

The actual call to `fmincon` is still more complicated in that it permits options to specify nonlinear constraints and to pass the model parameters to the criterion function. For the first version of the optimal portfolio model this function call is:

```
[x,fval,exitflag,output]=
    fmincon(@dcri1,x0,A1,b1,Aeq,beq,lb,ub,nonlcon,options,
                        beta,N,mu,sigma);
```

where

`exitflag`	=	provides info on the condition of the function at exit
`output`	=	provides additional output from the function
`dcri1`	=	the user supplied function that returns the criterion value for the first version of the model
`x0`	=	a vector of starting values to be used in the search for the optimal value of the function
`A1`	=	the matrix for the linear inequalities `A1 x <= b1` for the first version of the model
`b1`	=	the matrix for the linear inequalities `A1 x <= b1` for the first version of the model
`nonlcon`	=	specification for the nonlinear constraints
`options`	=	options to pass to the function
`beta, N, mu, sigma`	=	additional arguments to be passed to the function

For the second version of the model, where we minimize the variance subject to a constraint on the portfolio return, the call to the fmincon function is identical to the one above except that the user-supplied function is named `dcri2` and the matrix and vector for the set of linear inequalities are designated, respectively, as `A2` and `b2`.

The **MATLAB** code for the optimal portfolio model, which was programmed by Miwa Hattori, is shown in Appendix C and is also available in the book web site under the name portfolio.m. Other than the call to the function fmincon, the rest of the code is devoted primarily to preparing the inputs to pass to the function and to providing the function that returns the criterion value. Let us begin, then, with the code to pass the parameters θ and β and the number of equities in the portfolio, N, which is written

```
theta=10;
beta=2;
N=3;
```

The next section of the code, used to input the values of the mean-return vector μ and the covariance matrix Σ, is

```
mu=[8; 12; 15];
sigma=[6 -5 4;
-5 17 -11;
4 -11 24];
```

Note that each line of the vector mu ends with a semicolon, so mu is input as a column vector.

The next step is to provide the starting values that are to be used in the search for the optimum shares in the portfolio. A reasonable starting point is to divide the portfolio equally among the three equities. This is accomplished in the **MATLAB** code with the statements

```
X0=ones(N,1)/N;
Y0=ones(N,1)/N;
```

where

`x0` = the vector of starting values to be used in the search
for the optimal value of the function in the first version of the model

`y0` = the vector of starting values to be used in the search
for the optimal value of the function in the second version of the model

The **MATLAB** function `ones(m,n)` is used to create an n $\times$ m matrix of ones, and in this case the function call `ones(N,1)` creates an `N` vector of ones. All of the elements of this vector are then divided by `N`, so here with three equities the

vector `x0` has three elements, all of which are 0.33. The same is also true for `y0`, which is used with the second version of the model.

Next consider the linear inequality constraints for the two versions of the model. The first version has only a linear equality constraint and no linear inequality constraints so this is input with the **MATLAB** statements

```
A1=[];
b1=[];
```

that is, the matrix `A1` and the vector `b1` are empty and can be ignored by the function. However, this is not the case in the second version of the model, which minimizes the weighted variance subject to achieving at least a minimum mean return on the portfolio, that is,

$$\mu' y \geqslant \theta$$

However, since **MATLAB** expects the inequality in less-than-or-equal form it is necessary to multiply the constraint through by –1 to obtain

$$-\mu' y \leqslant -\theta \tag{14}$$

Then the `A2` matrix and the `b2` vector for this constraint can be input to the code with the statements

```
A2 = -mu';
b2 = -theta;
```

Since the vector mu was input above as a column vector it must be transposed with the transpose operator ($'$) here since we need it in the form of a row vector for this constraint.

The equality constraints for the two versions of the model are the same and are of the form

$$\sum_{i \in I} x_i = 1$$

for the first version and

$$\sum_{i \in I} y_i = 1$$

for the second version. Thus the A matrix and b vector have the same structure for both versions of the model and can be input with the statements

```
Aeq = [1 1 1];
Beq = 1;
```

The lower bounds on the variables are used to enforce the nonnegativity constraints and there are no upper bounds, so the bounds for both versions of the model are specified with the statements

```
lb=[0;0;0];
ub=[];
```

The final part of the model specification is the nonlinear constraints, of which there are none, so this is written

```
nonlcon=[];
options = optimset('MaxIter',60);
```

In addition, the options variable is used to set the maximum number of iterations for the nonlinear programming code to sixty. If the code has difficulty converging on the solution to your model it would be useful to raise this limit.

With all this preparation done, one can now call the fmincon function for the first versions of the model and print the key results with the statements

```
[x,fval,exitflag,output]
= fmincon(@dcri1,x0,A1,b1,Aeq,beq,lb,ub,nonlcon,options,beta,
          N,mu,sigma);
x
fval
```


This **MATLAB** function in turn calls the user-specified dcri1 function for the first version of the model to obtain the value of the criterion function at each point x in the search for the optimum. Recall that for the first version of the model the criterion function in matrix form is

$$J = \mu' x - \frac{1}{2}\beta x' \Sigma x \tag{15}$$

which can be written in index form as

$$J = \sum_{i \in I} \mu_i x_i - \frac{1}{2}\beta \sum_{i \in I} \sum_{j \in J} x_i \sigma_{ij} x_j \tag{16}$$

This can be rearranged slightly by moving the β and x_i to obtain

$$J = \sum_{i \in I} \mu_i x_i - \frac{1}{2} \sum_{i \in I} x_i \left(\beta \sum_{j \in J} \sigma_{ij} x_j \right) \tag{17}$$

which is the form used in the following dcri1 function:

```
function [z] = dcri1(x,beta,N,mu,sigma)
z=0;
     for i=1:N;
                temp=0;
                for j=1:N;
                          temp=temp+BETA*sigma(i,j)*x(j);
                end;
                z=z+mu(i)*x(i)-0.5*x(i)*temp;
     end;
z=-z;
```

Note at the top of the function that the `fmincon` function passes to the function `dcri1` the current point x and the parameters of the problem.

Note at the bottom of the function that the negative value of `z` is returned by the function. The reason is that `fmincon` –as its name indicates – is used to find the minimum value of a function. Therefore to use it to find the maximum, as we do here, it is necessary to reverse the sign of the criterion value.

An equivalent but more compact formulation that shows the power of **MATLAB** for matrix computation is

```
function z = dcri1(x,beta,N,mu,sigma) ;
z = -(mu'*x - 0.5*beta*x'*sigma*x);
```

The call to the `fmincon` function for the second version of the model, followed by the command lines to print the results, is

```
[y,fval,exitflag,output]
      =fmincon(@dcri2,y0,A2,b2,Aeq,beq,lb,ub,nonlcon,options,beta,
               N,mu,sigma);
y
fval
```

The `dcri2` function for the second version of the model is similar to `dcri1` but simpler since it does not contain the mu parameters:

$$J = \frac{1}{2}\beta y' \Sigma y$$

Moreover, it is not necessary to use the negative sign at the bottom of the function since in this case we are seeking a minimum.

9 Experiments

The logical experiment to do with the Markowitz model is to change the β risk preference parameter to see how the optimal portfolio changes. As β increases one would expect the optimal portfolio to contain larger proportions of stocks with lower variances and –most likely– with lower mean returns.

Another useful experiment is to change the pattern of the signs of the off-diagonal elements in the Σ matrix. In the original version used in this chapter there is a mixture of positive and negative off-diagonal elements. Selectively changing the signs of these elements and observing the results would be an interesting experiment.

Finally, another useful experiment is to compare the outcomes of the Monte Carlo code against those obtained with the optimization function. You may want to increase the number of time periods and/or the population size or change the value of the constant const and see how these changes affect the outcome of the code. Also, in the *random generation of new portfolios* section 7, you may want to divide the constant const by the number of runs index `k` and see how this affects the convergence path of the best weights to the optimal portfolio.

Of course you might want to obtain data on a set of stocks and bonds that are of particular interest and thus develop a personal version of the optimal portfolio model.

10 Further Reading

For a variety of financial models in **MATLAB** see Brandimarte (2001).

References

Brandimarte, P.: 2001, *Numerical Methods in Finance: A MATLAB-Based Introduction*, John Wiley, New York, USA.

Judd, K. L.: 1998, *Numerical methods in economics*, MIT Press, Cambridge, Massachussets.

Markowitz, H.: 1952, Portfolio selection, *Journal of Finance* **7**, 77–91.

Miranda, M. J. and Fackler, P. L.: 2002, *Applied Computational Economics and Finance*, MIT Press, Cambridge, Massachussets.

Appendix

A MATLAB Code for Optimal Portfolio Problem

MATLAB Code for a Markowitz Optimal Portfolio Problem

```
% Title: Quadratic-Linear Programming for Mean Variance Portfolio
% Analysis
% Program name: portfolio.m
% by Miwa Hattori
% Implementation of the mean-variance portfolio selection models
% with two alternative formulations in Matlab:
% maximizing expected mean return, net of variance costs and
% minimizing the overall variance costs of portfolio.

clear all;

%
% Preliminaries
%

theta=10; % Minimum mean-return on portfolio under formulation 2.

beta=2; % Subjective weight on returns variance of equities.

N=3; % Number of available equity types.

mu=[8; 12; 15];

% Column vector of mean annual returns on equities 1
% through N (%).

sigma=[6 -5 4;% Table of covariances between returns on equities.

-5 17 -11;

4 -11 24];

%
% Provide initial "guesses" for portfolio vectors.
%

x0=ones(N,1)/N; % Column vector of fractions of portfolio invested in

% equity i, initialized to 1/N.
```

```
y0=ones(N,1)/N; % Column vector of fractions of portfolio invested in

% equity i, initialized to 1/N.
%
% Constraints for optimization
% Matlab only has a function that solves a constrained nonlinear
% MINIMIZATION problem. See Help file for function "fmincon".
% fmincon finds a minimum of a multivariable function f(x)subject to
% A*x$<$= b, Aeq*x= beq, lb$<$= x $<$=ub where x, b, beq, lb, and
% ub are vectors, A and Aeq are murices.
%

A1=[]; % Set of linear inequality constraints under formulation 1.

b1=[];

A2=-mu'; % Set of linear inequality constraints under formulation 2:

b2=-theta; % Desired minimum mean-return on portfolio y $>$= theta (%).

Aeq=[1 1 1]; % Set of linear equality constraints.

beq=1; % Fractions x(i) must add to 1, fractions y(i) must add to 1.

lb=[0;0;0]; % Non negativity constraints on x(i) and y(i)

ub=[];

nonlcon=[]; % Non linear constraints-- none in this problem.

options=optimset('MaxIter',60);

%
% Definition of the criterion functions
% Functions dcri1, dcri2 are called. See files dcri1.m, dcri2.m.
%

[x,fval,exitflag,output]=fmincon(@dcri1,x0,A1,b1,Aeq,beq,lb,ub,nonlcon,options,beta,N,mu,sigma);

x

fval

[y,fval,exitflag,output]=fmincon(@dcri2,y0,A2,b2,Aeq,beq,lb,ub,nonlcon,options,beta,N,mu,sigma);

y

fval
```

```
% Title: Quadratic-Linear Progr for Mean Variance Portfolio Analysis
% Function Name: dcri1.m
% by Miwa Hattori
% The first formulation of the crit function for mean-variance port
% selection model.
% Defines the expected mean return, net of variance costs, which is
% to be maximized.

function z = dcri1(x,beta,N,mu,sigma);

z=0;

for i=1:N;

temp=0;

for j=1:N;

temp = temp + beta*sigma(i,j)*x(j);

end;

z = z + mu(i)*x(i) - 0.5*x(i)*temp;

end;

z=-z;

% Matlab only has a subrou to solve constrained MINIMIZATION problems.
% We solve a maximization problem by minimizing the negative of the
% objective function.
% Title: Quadratic-Linear Programming for Mean Variance Portfolio
% Analysis
% Function Name: dcri2.m
% by Miwa Hattori
% The second formulation of the criterion function for mean-variance
% portfolio selection model.
% Defines the overall variance costs of portfolio to be minimized.

function z = dcri2(y,beta,N,mu,sigma);

z=0;

for i=1:N;

temp=0;

for j=1:N;
```

```
temp = temp + beta*sigma(i,j)*y(j);

end;

z = z + 0.5*y(i)*temp;

end;
```

B Running MATLAB

This appendix provides the details for running the **MATLAB** software on a PC to solve the portfolio model. In order to use **MATLAB** with other input files, substitute the appropriate file name for `mcportfol.m` in the following [3].

- Go to the book web site at http://www.eco.utexas.edu/compeco and to the Input Files for Chapters in the Book section of the web site. Click on the `mcportfol.m` file name and select the *Save Target As* ... option in order to save the file in your preferred directory.
- Chose Programs from the Start menu and then choose **MATLAB**.
- In the Current Directory section of the main **MATLAB** window click on the icon that contains ... in order to browse to the folder where you stored the `mcportfol.m` file. Then double click on the `mcportfol.m` file name.
- A window that contains the `mcportfol.m` file opens. In order to solve the model pull down the Debug menu and select the Run option. A graph appears showing the results of the runs.
- In order to see the numerical results select the MATLAB main window and look in the Command Window section.
- If you run a **MATLAB** program that uses a number of functions stored in separate files (such as the portfolio.m or the models in the genetic algorithm chapters or in the agent-based model chapter) make sure you download all those files in the same directory.

[3] For help and information about obtaining **MATLAB** go to The MathWorks web site at http://www.mathworks.com.

C Running Mathematica

Mathematica is a widely available commercial software system. A web site for information about it is [http://www.wolfram.com].

Choose Programs from the Start menu and then choose Mathematica. Wait for a few seconds until a new window with a menu bar in its upper part appears. The content of this window is a white sheet called Notebook, which is like a document in a standard word processor. If you specified a file when opening Mathematica, this file is displayed.

Select Getting Started from the Help menu (located in the upper-right corner of the Mathematica window) and read the information that appears in the Info dialog box. To begin practicing with Mathematica, perform the calculations suggested in the section Doing Calculations. While doing this, you will appreciate the basic way of working with Mathematica, that is, your Notebook successively displays your inputs and the corresponding outputs. You may also notice that on the right side of your Notebook, a hierarchy of brackets appears. Each of them defines a cell (or a group of cells) that is the basic unit of organization in a Notebook. As you will quickly realize, cells can be hierarchically arranged (as in set and subsets). There are different kinds of cells: they can contain text, Mathematica input, Mathematica output, or graphs, and so on. Different small characters within each bracket identify the kind of cell. Here are some basic things you can do with cells:

- To edit a cell (i.e., to be able to work with it) just click on its bracket;
- To edit a group of cells, just click and drag on their brackets;
- To find out or change the kind of cell, edit the cell, select Style in the main menu and choose your option.
- To divide or merge cells, take the cursor to the division/insertion point of your choice, select Cell in the main menu and choose your options.
- To run a portion of a program contained in a cell of group of cells, just edit the cells and select Action-Evaluate in the main menu (or just press Shift-Enter)
- Finally, to save your Notebook, select File in the main menu and choose your options.
- Go to the book web site at [http://www.eco.utexas.edu/compeco] and then to the *Input Files for Chapters in the Book* section of the web site. Right click on the Leontief.nb file name and select the Save Target As ... option in order to save the file in your preferred directory.

- Go to the File menu and click Open to open the file**.**
- To run an input command or a cell containing a series of commands, click on the bracket on the right of it and hit Shift + Return (hold the shift key and hit Return at the same time). The output is displayed following the input, *unless* there is a **;** at the end of the input command line (**;** suppresses the output). You can run multiple cells by highlighting the corresponding brackets with your mouse and hitting Shift + Return once.
- Modify commands and rerun them sequentially, cell after cell, so that you can see the changes in the corresponding outputs.
- If you either select the outermost bracket and press Shift-Enter or go to the Kernel menu, choose Evaluation**,** and Evaluate Notebook**,** you rerun the complete program. If your program is large this may take a few minutes and it may be difficult for you to track down the results of your modifications. On the other hand, sometimes your modifications may require an updating of previous results, a clearing of previous values, or a change of attributes (and the Clear command or the SetAttributes command are usually at the beginning of the program). In these cases you may need to rerun the complete program to avoid errors or spurious results.

D Nonlinear Optimization Solvers

Solving nonlinear optimization problems usually requires the use of numerical methods. In general, those methods consist of a *smart* trial-and-error algorithm that is a finite sequence of computational steps designed to look for convergence to a solution. There is a variety of algorithms to solve nonlinear problems. Some of them are global methods, in the sense that they perform a parallel exploration of many regions of the optimization space, for example, the genetic algorithm. Other are local, as they tend to focus on the exploration of a particular region of the optimization space. We introduce here two of the most popular local methods—the gradient and the Newton methods—used by the solvers in Excel, **GAMS**, and **MATLAB**, but before introducing them, we give with a simple example.

Suppose that we are trying to find the maximum of a nonlinear function

$$y = f(x) \tag{D-1}$$

such as the one represented in Figure 2.

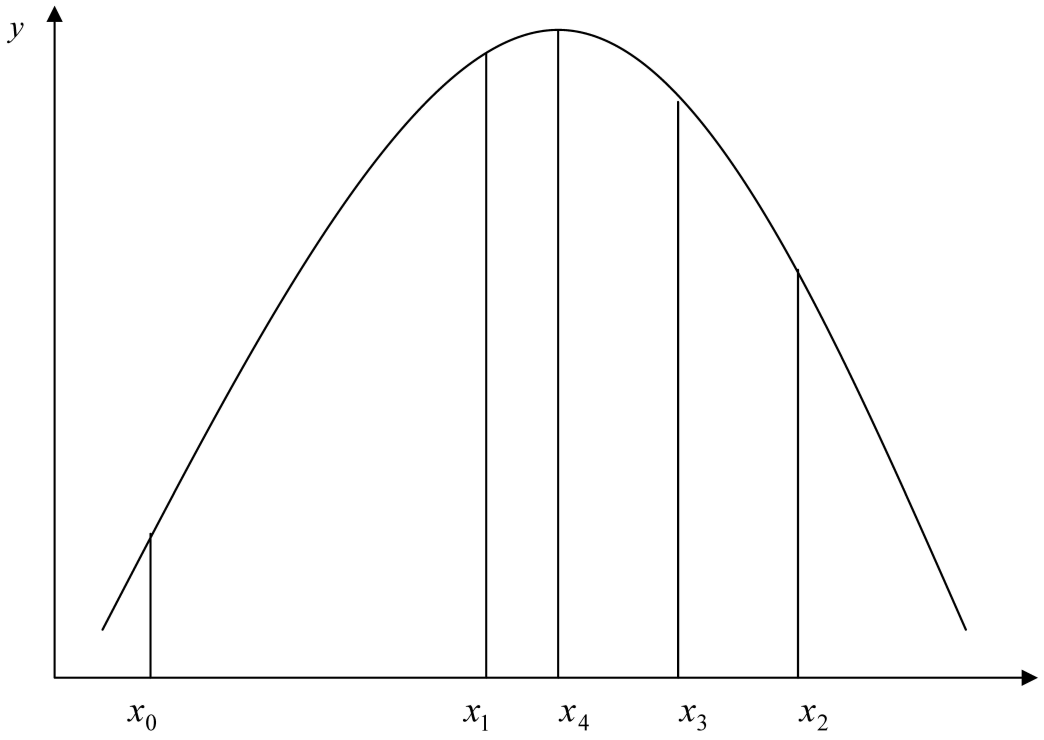
A simple and very rudimentary algorithm to find the solution might be as follows: We choose an arbitrary initial value for x, such as x_0 in Figure 2, and compute the corresponding $y_0 = f(x_0)$. We then increase that value by a constant magnitude h (we name this magnitude the *search step*) that we also choose in an arbitrary way. For the new value of x, that is x_1, we compute the corresponding value of

$$y_1 = f(x_1) = f(x_0 + h) \tag{D-2}$$

and compare this value to the one obtained in the previous step. We continue to do this as long as the differences between two successive values of y are positive (negative for a minimization problem). As soon as we compute a difference with a negative sign (in Figure F.1 this would correspond to x_2), we reverse the direction of the search. We begin to move in the opposite direction along x (i.e., subtracting h from x) and use a smaller value for h than the one we were using while we moved in the opposite direction. We continue like this until we again find a difference between two successive values of y that is negative, at which point we again reverse the direction of the search and further reduce the size of h, and so on. We stop when the difference between two successive values of y falls below a preestablished tolerance limit.

The gradient and the Newton methods are iterative like the one just presented.

Figure 2: A nonlinear function.



However, they exploit local information about the form of the function; that is, they use the function's derivatives. To illustrate this we change to a multivariate example. In this case we use the following equation to obtain each new value of the vector x:

$$x_{n+1} = x_n + h\Delta x \tag{D-3}$$

where h is the search step—now always a positive value—and Δx is the direction of change, which, as we will see, is determined by the function's derivatives.

The gradient method uses the first derivative or gradient, which gives us information about how the function changes in the neighborhood of a given point. Its basic framework is the well-known first-order Taylor approximation,

$$f(x_{n+1}) \cong f(x_n) + h\nabla f(x_n)\Delta x \tag{D-4}$$

where $\nabla f(x_n)$ is the gradient vector. Note that since h is supposed to be positive, the best direction of motion is

$$\Delta x = \nabla f(x_n) \tag{D-5}$$

for a maximization problem, since

$$f(x_{n+1}) \cong f(x_n) + h\nabla(f(x_n))^2 > f(x_n) \tag{D-6}$$

In addition, for a minimization problem

$$\Delta x = -\nabla f(x_0) \tag{D-7}$$

since

$$f(x_{n+1}) \cong f(x_n) - h\nabla(f(x_n))^2 < f(x_n) \tag{D-8}$$

The basic framework of the Newton method is the second-order Taylor approximation

$$f(x_{n+1}) \cong f(x_n) + h\nabla f(x_n)\Delta x + \frac{h}{2}\Delta x' H(x_n)\Delta x \tag{D-9}$$

where $H(x_0)$ is the second-order derivative or Hessian, which tells us how the slope of the function changes in a neighborhood of a given point.

Assuming the Taylor expansion of a function f is a good global approximation of that function, we approximate the optimum value of f by optimizing its

Taylor expansion. In our case, this is equivalent to saying that to determine the best direction of motion Δx we have to optimize the expression (f.9). Differentiating (f.9) with respect to Δx, setting the result equal to zero, and solving for Δx, we obtain

$$\Delta x = -\frac{\nabla f(x_n)}{H(x_n)} \tag{D-10}$$

which is the best direction of motion for the Newton method.

Sometimes iterative methods like the ones presented here do not converge to a solution after a finite number of iterations. This problem can be overcome by changing the maximum number of iterations, or the size of the search step, or the tolerance limit, or the initial value of the search. Most solvers allow you to change these parameters.

Note also, as is the general case for numerical methods dealing with nonlinear optimization problems, that if there is more than one local optimum we will find only one of them. Thus, we never know for sure if the optimum we reached was a local or a global one. A rough way of dealing with this difficulty is to solve the problem providing the algorithm with alternative initial values of the search.

In this appendix we presented three numerical methods of increasing complexity. Of course, the more complex ones make use of more information, thus reducing, in general, the number of steps needed to achieve convergence. However, those steps become more complex, since they require the computation of a gradient or a Hessian. Then there are trade-offs to be evaluated when choosing a solution method.

There are additional methods for solving nonlinear problems numerically—for example, the conjugate gradient method, the penalty function method, and the sequential quadratic programming—a number of which extend, combine, or mimic the ones introduced here. For a comprehensive presentation refer to Judd (1998) and Miranda and Fackler (2002). The Excel solver uses a conjugate gradient method or a Newton method. **GAMS** uses a variety of methods, depending on what you choose or have set up as the default nonlinear solver. The **MATLAB** solver used in Chapter 7 and invoked by the fmincon function uses a sequential quadratic programming method. For details on the specific methods used by **Excel**, **GAMS**, and **MATLAB** refer to their corresponding user's and solver's manuals.

E GAMS Code for a Markowitz Optimal Portfolio Problem

The complete GAMS version of the model, which was programmed by Seung-Rae Kim, is at the end of this appendix. Here we discuss the parts of the model. The first part of the GAMS statement of the model is the specification for the set of equities:

```
Set i equities /equity1, equity2, equity3/;
```

While it is more common in GAMS to use an upper case letter for the set so that the specification would be Set I instead of Set i there is an argument for using the lower case specification as we do here. The argument is that in GAMS the symbols for sets are used where the mathematics of the model would indicate a set *and* where the mathematics would indicate an index. Thus the mathematical statement of equation (7), that is,

$$\sum_{i \in I} x_i = 1$$

is written in GAMS as

```
sum(i, x(i)) =e= 1.0 ;
```


Of course, since GAMS does not distinguish between upper and lower-case letters it would be possible to write the GAMS statement as

```
sum(I, x(i)) =e= 1.0 ;
```

This might be more aesthetically pleasing but could also be more confusing. Just beneath the set specification statement in GAMS is an Alias statement of the form

```
Alias (i,j) ;
```

This statement creates a set J which is a copy of the set I. This kind of statement is used in GAMS when there is a double summation over the elements of a variable x(i,j) of the sort that is used in computing the variance of the portfolio in this model.

Next the data are input using the Scalar keyword for θ andβ, the Parameters keyword for the vector μ, and the Table keyword for the matrix Σ as follows:

```
Scalar theta desired minimum mean-return on portfolio ({\%}) / 10 /
beta subjective weight on returns variance of equities / 2 /;
Parameters mu(i) mean annual returns on equities ({%})
/ equity1 8
equity2 12
equity3 15 / ;

Table sigma(i,j) covariance matrix of returns on equities
equity1 equity2 equity3
equity1 6 -5 4
equity2 -5 17 -11
equity3 4 -11 24 ;
\end{footnotesize}
```

Then the variables are defined using the keyword *Variables*

```
Variables

x(i) fraction of portfolio invested in equity i in formulation 1
y(i) fraction of portfolio invested in equity i in formulation 2

criterion1 expected mean return on portfolio, net of variance cost
criterion2 variance-augmented total risk cost of portfolio ;

Positive Variable x, y ;
```

Also, the Positive Variable statement is used in GAMS to enforce the non-negativity constraints on the x and y variables. Note that the two versions of the model using the x variables in the one version and the y variables in the other are being developed simultaneously in the GAMS statement of the models, rather than one after another.

Next comes the declaration of the equations with the statements

```
Equations dcri1 definition of criterion1

dcri2 definition of criterion2

xsum fractions x(i) must add to 1.0

dmu desired minimum mean-return on portfolio y(i)

ysum fractions y(i) must add to 1.0 ;
```

Recall that the semicolon after the last line above is crucial in GAMS. It is easy to forget this semicolon when developing a model; however, forgetting it usually results in errors in GAMS, since the compiler does not know where the

list of equation names ends and the definition of the equations begins.

Next are the definitions of the equations beginning with the criterion function. However, since GAMS uses index rather than matrix notation, it is useful to restate the matrix form of the criterion function from equation (6), that is,

$$J = \mu' x - \frac{1}{2}\beta x' \Sigma x$$

in index form as

$$J = \sum_{i \in I} \mu_i x_i - \frac{1}{2}\beta \sum_{i \in I} \sum_{j \in J} x_i \sigma_{ij} x_j$$

where

$$\begin{aligned} \mu_i &= \text{the mean return on equity } i \\ \sigma_{ij} &= \text{the covariance of the returns on equities } i \text{ and } j \end{aligned}$$

This criterion is written in GAMS as

```
dcri1.. criterion1 =e= sum(i, mu(i)*x(i))
   -.5*sum(i, x(i)*sum(j, beta*sigma(i,j)*x(j))) ;
```

Here we see the use of the alias I and J sets for the double summation. Similarly the criterion function for the second version of the model, which is written in matrix form as

$$J = \frac{1}{2}\beta y' \Sigma y$$

becomes

```
dcri2.. criterion2 =e= .5*sum(i, y(i)*sum(j, beta*sigma(i,j)*y(j))) ;
```

in the GAMS statement. The rest of the constraints for the two versions of the model are stated in GAMS as

```
xsum.. sum(i, x(i)) =e= 1.0 ;
dmu.. sum(i, mu(i)* y(i)) =g= theta ;
ysum.. sum(i, y(i)) =e= 1.0 ;
```

The first and third constraints above require that the fractional portfolio holdings add to 1. The middle constraint containing the θ parameter is the restriction on the portfolio return in the second version of the model.

The last part of the GAMS statement for the two versions of the model is

```
Model portfolio1 / dcri1, xsum / ;
Model portfolio2 / dcri2, dmu, ysum / ;
Solve portfolio1 using nlp maximizing criterion1;
Solve portfolio2 using nlp minimizing criterion2;
```

Here we see a good example in GAMS of the use of Model statements to specify different versions of a model that can then be solved one after another with two different Solve statements. The two models are actually quadratic programming models; however, GAMS does not have a specialized solver for this purpose and the nonlinear programming solver called by the keyword nlp is appropriate. For an introduction to this type of solver see Appendix D.

Below is the complete GAMS code for the Markowitz problem. The GAMS library has a variety of optimal portfolio models which may be of interest to the reader. They are called PORT and QP1 through QP6. The PORT model was created by the Control Data Corporation and the QP models were created by Erwin Kalvelagen at the GAMS Corporation:

```
$Title A Quadratic-Linear Program for Mean-Variance Portfolio Analysis
$Ontext
* Program by Seung-Rae Kim
These are mean-variance portfolio selection models with two alternatives
formulations in GAMS: (1) maximizing expected mean return, net of
variance costs, & (2) minimizing the overall variance costs of portfolio.
$Offtext

 Set i equities /equity1, equity2, equity3/;
 Alias (i,j) ;
 Scalar theta desired minimum mean-return on portfolio (%) / 10 /
        beta subjective weight on returns variance of equities / 2 /;

Parameters mu(i) mean annual returns on equities (%)
 / equity1  8
   equity2 12
   equity3 15 / ;

 Table sigma(i,j) covariance matrix of returns on equities
         equity1 equity2 equity3
 equity1    6     -5       4
 equity2   -5     17     -11
 equity3    4    -11      24 ;

 Variables
 x(i) fraction of portfolio invested in equity i in formulation 1
 y(i) fraction of portfolio invested in equity i in formulation 2
 criterion1 expected mean return on portfolio, net of variance cost
 criterion2 variance-augmented total risk cost of portfolio ;
```

```
Positive Variable x, y ;

Equations dcri1 definition of criterion1

dcri2 definition of criterion2
xsum fractions x(i) must add to 1.0
dmu desired minimum mean-return on portfolio y(i)
ysum fractions y(i) must add to 1.0 ;
dcri1.. criterion1 =e= sum(i, mu(i)*x(i))-.5*sum(i, x(i)*sum(j, beta*sigma(i,j)*x(j))) ;
dcri2.. criterion2 =e= .5*sum(i, y(i)*sum(j, beta*sigma(i,j)*y(j))) ;
xsum.. sum(i, x(i)) =e= 1.0 ;
dmu.. sum(i, mu(i)*y(i)) =g= theta ;
ysum.. sum(i, y(i)) =e= 1.0 ;

Model portfolio1 / dcri1, xsum / ;
Model portfolio2 / dcri2, dmu, ysum / ;

Solve portfolio1 using nlp maximizing criterion1;
Solve portfolio2 using nlp minimizing criterion2;
```