Retirement Income Analysis
with scenario matrices

William F. Sharpe

2. Scenario Matrices

As the title indicates, this book is concerned with analyses of different types of strategies for the provision of income during peoples' retirement years. And many of these analyses will utilize *scenario matrices,* which will be defined and discussed in this chapter.

The main analytic tools employed in the book are those of economics, mathematics and computer programming. The mathematics will rely heavily on matrix algebra. And the computer programming algorithms will be expressed in programming language syntax. This chapter provides fundamental information for both ingredients.

Probabilistic Forecasts

Consider a game in which someone will flip a coin two times. Each time the coin can land showing “heads” (H) or “tails” (T). The four possible outcomes can be represented in a table with four rows and two columns, as follows:

	Time 1	**Time 2**
Scenario 1	H	H
Scenario 2	H	T
Scenario 3	T	H
Scenario 4	T	T

Each row in this table represents a possible future multi-period outcome, which we will call a *scenario*. One and only one of these outcomes will occur in the future, but we don't know in advance which one it will be. The columns in the table represent sequential times in the future at which events will occur. Here, the body of the this table has four rows and two columns, since there are four scenarios and two time periods.

We will term the body of the table a *scenario matrix*. In this case:

H H
H T
T H
T T

This matrix has four rows and two columns, so its size is 4 by 2 (sometimes written as 4x2). We will adopt the convention that in all such matrices, the rows represent scenarios and the columns represent time periods (in sequence).

Our representation of the coin flipping game is not complete. To be perfectly general we need to specify the probabilities that the various scenarios will actually occur. If the coin is "fair", then there is 1 chance out of 4 that the outcome will be the first scenario, 1 chance out of four that it will be the second scenario, and so on. Conventionally, probabilities are stated as the proportion of times an outcome would be achieved in many repeated trials of the process. Here, the probabilities are all equal to 0.25.

	Probability
Scenario 1	0.25
Scenario 2	0.25
Scenario 3	0.25
Scenario 4	0.25

This set of probabilities can be considered a matrix with four rows and one column (4x1). Alternatively it may be called a *vector* – a term that denotes a matrix with only one column or one row. More specifically, this could be termed a four-element *column vector*. However, it is simpler to use the more general term, *matrix* for tables with multiple rows and columns, those with only one column, and those with only one row, specifying the number of rows and columns whenever needed. One could go even farther, denoting a single value as a *1x1 matrix* but in most cases this would seem to be an affectation.

Note that in this case the set of outcomes can be described efficiently by specifying the scenario matrix and indicating that each row (scenario) is equally probable. However, that might not be the case. Consider a situation in which the coin is not "fair" and has been shown to end up showing heads 6 times out of every 10 flips. Then the probability of a head in a given flip is 0.60 and that of a tail is 0.40. This implies the following probabilities:

Time 1	**Time 2**	**Probability**
H	H	0.60 * 0.60 = 0.36
H	T	0.60 * 0.40 = 0.24
T	H	0.40 * 0.60 = 0.24
T	T	0.40 * 0.40 = 0.16

In this case, the scenario matrix is not sufficient; one also needs the probability matrix (vector).

Representative Scenarios

As we have seen, in the case in which a fair coin is flipped repeatedly, only a scenario matrix is needed since each scenario (row) is equally probable. If there are four scenarios, the probability of any one occurring is ¼. If there are *n* scenarios, each has a probability of occurrence of *1/n*. The scenario matrix contains all the relevant information about possible future outcomes. But this only suffices for the special case in which the coin is fair. This may be a good assumption for most coins, but in most practical problems, the probabilities for different scenario will differ.

Moreover, there may be a great many possible scenarios. Imagine a game in which the outcome in each time period is the change in the value of an index of common stocks. There are a great many possible outcomes for any single time period, and the number of possible outcomes over multiple time periods will be much, much greater. Many analyses of financial problems assume continuous probability distributions of investment returns over a single time period, and some even consider continuous time. When feasible, analytic approaches have substantial advantages, but the associated mathematics can be daunting and make it very difficult, if not impossible, to use such methods to analyze multi-period strategies that involve complex reactions to changes in financial values and other variables.

This book takes a different approach, simplifying problems to focus on a discrete number of future time periods and possible multi-period scenarios. To illustrate, consider the previous case in which our coin has a 0.60 probability of coming up heads. Imagine flipping it twice, recording the outcome (e.g. H T), then repeating the experiment 99,999 times, recording each outcome in a separate row of a matrix. This would generate a (100,000 x 2) matrix of possible scenarios. Moreover, it would seem perfectly sensible to assume that each of these scenarios has a probability of 1/100,000 of occurring. Why? Because in all likelihood the proportion of scenarios with two heads will be very close to 36%, the proportion with two tails will be very close to 16%, and so on.

Now consider what could be done with a computer. There is no need to actually flip a physical coin for this analysis, once its properties have been determined (in this case the likelihoods that it will come up heads or tails). A computer can easily produce the desired scenario matrix. One writes a *program,* implementing a procedure (*algorithm*), or a series of such programs, in a suitable computer programming language, then lets the computer do the hard work. And, after a scenario matrix has been produced, another program can analyze the properties of the underlying strategy. This book will provide many examples of such an approach.

Monte Carlo Analysis

The use of representative scenarios may seem a complicated procedure, wasteful of both computer time and information storage. For a game in which a single coin is flipped twice, it is. But computer time and memory are cheap. And computer algorithms, once determined to be correctly programmed, are consistent and reliable. With rare exceptions, this is not easily said of most humans.

The idea of flipping a coin twice, then repeating the exercise 99,999 times is, of course, ludicrous. Instead one asks a computer to do the experiment. This process is generally termed *Monte Carlo Analysis*, since it can be considered the simulation of a game that might be featured at the famous casino in Monte Carlo, Monaco. The term seems suitable for a coin-flipping game, but hardly so for some of the other sources of uncertainty that affect retirement income. Hence we will use this terminology sparingly.

Consider the case in which the probability distribution of outcomes for a coin flip is:

Heads: 0.60
Tails: 0.40

To make things simple, let the number 0 denote a head and the number 1 a tail. Now imagine that there were a computer process that could generate a random number between 0 and 1 whenever requested, with each value in the range from 0 to 1 equally likely.

Imagine that you could ask the computer to produce a matrix with 10 rows and 2 columns of such random numbers by typing:

```
>> z = rand(10,2)
```

And that this would produce:

```
z =
   0.7958   0.4069
   0.2667   0.0621
   0.3320   0.2312
   0.3704   0.4271
   0.4052   0.3936
   0.0920   0.6750
   0.2562   0.9791
   0.2631   0.7870
   0.2671   0.8025
   0.4676   0.3690
```

Now imagine that we could ask which numbers are greater than 0.60, with 1 representing an affirmative answer and 0 a negative answer, by typing:

```
>> m  = (z > 0.60)
```

producing:

```
m =
     1     0
     0     0
     0     0
     0     0
     0     0
     0     1
     0     1
     0     1
     0     1
     0     0
```

We can interpret this as a scenario matrix (in which each row is a scenario, each column a time period), and the elements represent heads (0) or tails (1).

Clearly, matrix **m** is not completely representative of the underlying probabilities of heads and tails. In the first period (column 1) there are 9 heads (0's) while the probabilities would lead one to expect 6. In the second period (column 2) there are in fact 6 heads, but there could have been more or fewer. Looking at the rows as a whole, the results are even less representative. For example, there is no scenario in which a tail is followed by a tail, even though the probability of such an outcome is 0.16.

Not surprisingly, ten scenarios are not enough to be reasonably representative of the underlying probabilities of different outcomes. Nonetheless, by typing two very short instructions we were able to produce scenarios representing the true probabilities of different outcomes. And it would be a simple matter to change the first statement to:

```
>> z = rand(100000,2);
```

to generate 100,000 scenarios (but without listing them at the time, as indicated by the final semicolon).

MATLAB

The two statements in the preceding example were both simple and powerful. Happily, they were not in fact imaginary. Each could be typed as shown, after a command-line prompt (>>) in the command window for a language called MATLAB, producing matrices similar to the ones shown, although the actual contents would differ each time due to the generation of different random numbers. And a larger number of scenarios could be generated by changing the first command (although, as indicated, it would be desirable to finish each statement with a semicolon (;) to avoid having the results listed).

MATLAB is a computer language with a great many features including the ability to create and manipulate variables that are in fact matrices. It is a product of a company called the Mathworks, founded in 1984 to build on some of the techniques previously covered in an important book titled “Computer Solution of Linear Algebraic Systems” by George Forsyth and Clive Moler. Forsyth was a Stanford Professor; Moler one of his PhD students who subsequently founded Mathworks. The Forsyth/Moler book included algorithms for matrix operations written in general-purpose computer languages of the day (Algol, Fortran and PL/1). Subsequently, Moler and others developed matrix routines in other languages and, eventually, languages in which matrices were fundamental objects. MATLAB is one such language.

Since its origins, MATLAB has grown substantially. The name originally was shorthand for “Matrix Laboratory”. But now the company describes it as “The Language of Technical Computing”. Or, only slightly less grandly, “... a multi-paradigm numerical computing environment and fourth-generation programming language.” Moreover, Mathworks has developed many “toolboxes” (available for additional fees) which include functions designed for analyses in many fields, including mathematics, statistics, engineering and finance.

MATLAB has not only grown to include a great many powerful and sophisticated features, it also has enjoyed careful attention to detail, is extremely robust and (to use the industry term) close to bug-free. This is not without cost. Commercial and individual users must pay for the right to use the software – typically a one-time fee of more than $2,000 for a license to use it on up to three computers. No additional fees are required, but the annual fee for optional updates and other services is typically between $300 and $400. Toolboxes are available for additional fees. Corporate and other users may obtain multi-user licenses at varying costs.

Fortunately, the company provides MATLAB to universities, colleges and other non-profit organizations at reduced rates and offers students at such institutions individual licenses for use on their own computers or online for one-time fees as low as $50.

From here on, I assume that you will at least be willing to learn to read and understand MATLAB code. With luck, you will gain access to MATLAB, run some of the code in the book, and adapt it for use in your own research or practice.

A final comment along this line. MATLAB is not the only language that could be used for the types of analyses to be covered here. Another alternative would be the Python language augmented with the *NumPy* and *SciPy* libraries, all of which are open-source and free. These libraries implement a number of matrix operations and may (or may not) do so with the simplicity, reliability, efficiency and seamless memory management provided by MATLAB. Exploration of Python and other possible systems are left for others. Here, MATLAB will reign.

Computer Power

The approach advocated here requires, at the very least, the creation and storage of several very large matrices. Examples include matrices of market portfolio returns, inflation, present values of payments, incomes, recipients of income, and investment fees paid. In a typical application, each such matrix will have 100,000 or more rows (scenarios) and 50 or more columns (future years). This could require storage for as many as 5,000,000 (5 million) numbers for each matrix. Using the default 8-byte format for double-precision numeric values, the needed storage for a single matrix could thus be as large as 40 million bytes. If up to ten such matrices need to be available at a given time, 400 million bytes of storage could be needed. Moreover, to keep processing times within reasonable limits, it is necessary that information can be written to and retrieved from storage very quickly. And finally, the processor must run at a reasonably high speed to avoid excessively long processing times.

This may seem a tall order. But not with today's computer technology. 400 million bytes is equal to 0.40 gigabytes (since a gigabyte is 1 billion bytes). My somewhat venerable Macbook Pro laptop computer has 8 gigabytes of main memory. Moreover most operating systems can swap information in and out of "disk storage" as a last resort, if needed. This can be prohibitively slow using actual rotating disk media, but considerably quicker if remarkably cheap solid-state "flash" storage is used instead. My Macbook has 256 gigabytes of such storage, the majority of which is available for use if needed. Finally, modern processors are very fast. My computer has an Intel quad core i7 processor running at 2.3 gigahertz, which can process matrix operations in MATLAB remarkably quickly. In late 2014, a comparable new computer cost approximately $2,000 in the United States. And a machine with 4 times the storage (a terabyte equal to 1,000 gigabytes) and a faster processor (running at 2.8 gigahertz) cost roughly $3,000.

The economics are straightforward. Computer hardware continues to become cheaper and cheaper. Not so for human time, aspiration and patience. For the analysis of retirement income strategies, there is every reason to use matrices large enough to be sufficiently representative of the range of possible future scenarios and to write programs that can make procedures used to process such matrices concise, efficient and easily understood.

This book is intended primarily for those with access to an efficient modern computer with MATLAB installed or available online. If others will find it useful, so much the better.

The Scratch Retirement Income Scenario Program

Before proceeding, a short note is in order about another set of software that I have developed, which takes a very different approach but addresses similar issues.

The blog RetirementIncomeScenarios.blogspot.com uses a set of software written in the Scratch programming language, developed, maintained and supported by the Lifelong Kindergarten Group at the Massachusetts Institute of Technology Media Lab (scratch.mit.edu).

Scratch is designed to be taught to students between 8 and 16 years of age. I have used it to teach beginning programming to students in this age range and found it superb for the purpose. But Scratch has no built-in matrix operations. Moreover, it processes computations at relatively slow speeds, due in part to the need for programs to work well in most browsers and on many types of computer. As a result, my Scratch code for retirement income strategy analyses was difficult to write and is hard for others to read. Also, processing speed for the Scratch program limits the number of retirement income scenarios that can be analyzed in a reasonable time to 5,000 or fewer. And some types of analysis that will be covered here are simply infeasible to program in Scratch.

Think of my scratch blog and software as a kind of demonstration project for people interested in some of the ways in which retirement income can be analyzed . But this book is designed for people who wish to do serious analyses of retirement income strategies and/or understand in depth the manner in which analyses can be performed. If that is you, please read on.

Programming Manuals

This is not a book on MATLAB *per se;* many are now available. A concise introduction and reference is *How to write MATLAB commands -- A handbook guide to common syntax* by Derek Causon and Clive Mingham. It is available in electronic form (kindle) from Amazon for $0.99 (and free for some Amazon users). There are many comprehensive MATLAB books. And MATLAB itself is well documented internally and online.

MATLAB is an interpretive languages and has a very convenient desktop interface (Interactive Development Environment) that can have multiple windows. A *Command Window* can be used to enter statements to be executed immediately. A companion *Editor Window* can display one or more program files and can be used to change or save any such file at any time. Program files and functions can be executed by name from the command window or by another program. On this desktop, it is a simple matter to move between windows, doing experiments, changing files, and so on.

The MATLAB language and development environment can save users substantial amounts of programming time and reduce frustration with logical and other errors. They also produce programs that can execute very quickly, especially when matrix operations are involved. Versions are available for platforms as diverse as Microsoft Windows, Linux and Mac OS.

According to Wikipedia (accessed in November 2014), " MATLAB users come from various backgrounds of engineering, science, and economics. MATLAB is widely used in academic and research institutions as well as industrial enterprises." There are also users in many financial firms. And there might be more if this book achieves its goals.

Matrix Operations

Now to matrix operations. The following sections will describe a number. There is no need to memorize them; they will be described again when used. The goal here is to indicate the wide range of built-in and highly efficient operations that are available, and to introduce some of the tools that will be used in following chapters.

To start, we'll create a new matrix.

```
>> x = [1 2 3; 4 5 6]
x =
     1     2     3
     4     5     6
```

The command typed after the >> prompt) is an *assignment* statement. A variable name (x) is followed by an equal sign, to the right of which is the information to be assigned to the variable. Some would read this as “x gets [1 2 3 ; 4 5 6]”. Others would think of the equal sign as a left-pointing arrow. In effect, the expression to the right of the equal sign is evaluated, then the result assigned to the variable to the left of the equal sign.

In the command, the brackets denote a matrix. The semicolon separates the elements of the first row from those of the second row. In general, semicolons are used as separators. Inside the brackets, they separate elements of rows. If there are multiple statements on a line, semicolons can be used to separate them. And in an assignment statement such as this, a semicolon at the end could be used to suppress the printing of the result, but in this case the results are to be shown after the command is typed. The last three lines, produced by the system, show that x is a 2x3 matrix (with two rows and three columns), as intended.

Operations with Scalars

Now let's do some simple operations with a matrix and a scalar (that is, a single variable or constant).

```
>> y = x + 3
y =
    4    5    6
    7    8    9
```

As can be seen, a constant added to a matrix increases each element by the constant amount.

You can also subtract a constant from each element in a matrix, for example:

```
>> y = x - 2
y =
   -1    0    1
    2    3    4
```

And you can multiply or divide each element by a constant, as in:

```
>> y = x * 2
y =
    2    4    6
    8   10   12
```

and

```
>> y = x/2
y =
    0.5000    1.0000    1.5000
    2.0000    2.5000    3.0000
```

With all but the division operation, the order doesn't matter – either the matrix or the constant can be first.. And the scalar can be a numeric value, as in the examples, or a variable, as in:

```
>> v = 3.1
v =
    3.1000
>> y = x + v
y =
    4.1000    5.1000    6.1000
    7.1000    8.1000    9.1000
```

Element-wise Operations with Two Equal-sized Matrices

Now to operations involving two matrices of equal size – that is, with the same number of rows and the same number of columns. The idea is to produce a new matrix with each element based on the values of the elements in the same positions in the original matrices.

To add the elements in two such matrices we simply use the plus sign. For example:

```
x =
     1     2     3
     4     5     6
y =
     4     5     6
     7     8     9
>> z = x + y
z =
     5     7     9
    11    13    15
```

This also works for subtraction.

```
>> z = y - x
z =
     3     3     3
     3     3     3
```

Multiplication and division require a slightly different notation, since the standard symbols (* and /) are used for other matrix operations. Element-wise operations are invoked with the operators .* and ./ as in these examples:

```
>> z = x.*y
z =
     4    10    18
    28    40    54
>> z = x./y
z =
    0.2500    0.4000    0.5000
    0.5714    0.6250    0.6667
```

Once again – these operations only work if the matrices involved are of the same size – reason enough to insure that our key scenario matrices cover the same number of scenarios (rows) and years (columns).

Matrix Multiplication

Many of the procedures in this book use element-wise matrix operations. But some employ "true" matrix multiplication. This is best described with an example.

m1 =

1	**2**	**3**
4	**5**	**6**

m2 =

1	**2**	**3**	**4**
5	**6**	**7**	**8**
9	**10**	**11**	**12**

m3 = m1*m2

38	**44**	**50**	**56**
83	**98**	**113**	**128**

Note that m1 is a 2x3 matrix and that m2 is 3x4 matrix. Their product is 2x4 matrix. As in any such matrix multiplication, the number of columns in the first matrix must equal the number of rows in the second. And the number of rows in the product will equal the number in the first matrix while the number of columns will equal the number in the second matrix. More simply put, if you multiply an (r1 x Z) matrix by a (Z x c2) matrix the result will be an (r1 x c2) matrix. Here, the product of a (2x3) matrix and a (3x4) matrix is a (2x4) matrix. Perhaps not surprisingly, each element in the final matrix will be the sum of the products of the elements in the corresponding row of the first matrix times those in the corresponding column of the second. Thus, in our example, m3(1,1) = 1*1 + 2*5 + 3*9 = 38, and so on.

This may seem arcane, but we will frequently find such operations helpful, beginning in the next chapter.

Functions for Creating Matrices

It is hardly feasible to type in the contents of a matrix with 100,000 scenarios (rows) and 50 years (columns). Nor shall we. Fortunately there are a number of ways to create such matrices *de novo*.

We have already seen one approach that fills a matrix with random numbers between zero and one. Here is a simple example:

```
>> z = rand(2,3)
z =
   0.0266   0.7132   0.2896
   0.4283   0.8035   0.8800
```

Another will generate random values from a normally-distributed (bell-shaped) distribution with a mean of zero and a standard deviation of 1:

```
>> z = randn(2,3)
z =
  -0.3783   1.5106   0.5412
  -1.7094   0.0608  -0.0185
```

Of course for a serious scenario matrix, more values will be required and it is important to end the statement with a semicolon to avoid printing out all the entries. For example:

```
>> z = randn(100000,50);
```

On my computer this takes about 1/10 of a second. So don't worry about processing time.

There are other functions that can generate matrices *de novo*. For example:

```
>> z = ones(2,3)
z =
    1    1    1
    1    1    1

>> z = zeros(2,3)
z =
    0    0    0
    0    0    0
```

It is sometimes useful to start by setting up a matrix of a desired size using one of these functions, then to modify it as needed with other operations.

Transposing a Matrix

It is not unusual to wish that a matrix were oriented differently. For some operations, it would be much better if the rows were columns and the columns were rows. More simply put, it would be desirable to flip the matrix 90 degrees. Not to worry. There are in fact two ways to *transpose* a matrix.

The first uses a function:

```
x =
     1     2     3
     4     5     6

y = transpose(x)
y =
     1     4
     2     5
     3     6
```

The second does it with one keystroke (the apostrophe):

```
>> y = x'
y =
     1     4
     2     5
     3     6
```

Matrix Functions That Produce Vectors

This rather wonky title covers a number of very useful functions. Each operates on a matrix column by column, producing some value for each. The result is a row vector with the information requested.

Examples are better than words. All will use our simple matrix:

```
x =
   1   2   3
   4   5   6
```

The first function sums the values in each column:

```
>> y = sum(x)
y =
   5   7   9
```

The next takes the product of all the numbers in each column:

```
>> y = prod(x)
y =
   4  10  18
```

Want to find the minimum value in each column? Simple:

```
>> y = min(x)
y =
   1   2   3
```

And, of course, the maximum values:

```
>> y = max(x)
y =
   4   5   6
```

We have not forgotten statisticians. Here is the function used to find the arithmetic means of the values in each column:

```
>> y = mean(x)
y =
   2.5000    3.5000    4.5000
```

One for the standard deviations:

```
>> y = std(x)
y =
   2.1213    2.1213    2.1213
```

Another to compute the medians:

```
>> y = median(x)
y =
   2.5000    3.5000    4.5000
```

In this case the means and the medians are the same, since there are only two elements in each column. But with many scenarios in the rows, the values could be very different indeed.

The decision to have these functions operate column-by-column may seem to have been arbitrary, and it probably was. But it is a simple matter to have them do the computations row-by-row. Simply transpose the matrix, use the function, then transpose the result. For example:

```
>> x = [1 2 3; 4 5 6]
x =
    1    2    3
    4    5    6
>> y = sum(x')'
y =
    6
   15
```

Short, sweet and highly effective.

Matrix Functions That Produce New Matrices

We now turn to operations that use the data in one matrix to produce another.

The *cumsum* ("cumulative sum" operator produces a matrix in which each element equals the sum of the corresponding element in the original matrix plus the values of all the elements above it in the same column. For example:

```
x =
     1     2     3
     4     5     6
>> y = cumsum(x)
y =
     1     2     3
     5     7     9
```

The *cumprod* ("cumulative product") operator is similar, but produces a matrix in which each element is the product of the corresponding element in the original matrix times the values of the elements above it in the original matrix:

```
>> y = cumprod(x)
y =
     1     2     3
     4    10    18
```

We shall make considerable use of the next two functions, which produce new matrices in which each element is a function of the corresponding element in the original matrix. The first computes the natural logarithms (to base *e*):

```
>> y = log(x)
y =
         0    0.6931    1.0986
    1.3863    1.6094    1.7918
```

The next does the reverse, with each element in the new matrix equal to e raised to the power given by the corresponding element in the initial matrix:

```
>> y = exp(x)
y =
    2.7183    7.3891   20.0855
   54.5982  148.4132  403.4288
```

There are other functions that work with matrices in a similar manner, but these are the most important for our purposes.

The next function, *sort*, is extremely useful for producing probability distributions of scenario outcomes for each year. It produces a new matrix in which each column contains the same data as the corresponding column in the original matrix, but with the values sorted in either ascending or descending order. For example:

```
>> y = sort(x,'descend')
y =
    4    5    6
    1    2    3
```

Of course, you can write 'ascend' instead, with the predictable results.

There is another form of the sort operation, in which the results include the new sorted matrix and a matrix of the row numbers of each of the elements in the old matrix. For example:

```
>> x = rand(3,4)
x =
   0.8147   0.9134   0.2785   0.9649
   0.9058   0.6324   0.5469   0.1576
   0.1270   0.0975   0.9575   0.9706
>> [sort ii] = sort(x,'descend')
sort =
   0.9058   0.9134   0.9575   0.9706
   0.8147   0.6324   0.5469   0.9649
   0.1270   0.0975   0.2785   0.1576
ii =
    2    1    3    3
    1    2    2    1
    3    3    1    2
```

In most cases, the matrix of original locations (*ii*) is not needed, but there are a few situations where it can be very helpful.

It is also possible to apply any logical test to a matrix, producing a new matrix in which each cell indicates whether the value in the corresponding cell of the original matrix met the test (*1*, for true) or not (*0* for false).

Here is an example:

```
>> x = randn(2,3)
x =
    0.7254    0.7147   -0.1241
   -0.0631   -0.2050    1.4897

>> y = ( x > 0 )
y =
     1     1     0
     0     0     1
```

First, a word about the use of parentheses. In this example the parentheses around *x>0* could have been omitted. Some professional programmers take great pride in knowing the order in which parts of program statements are executed, and minimizing the use of parentheses accordingly. But many programs are written for both computers to process and for other people to read. I favor the use of parentheses to make the originator's intent as clear as possible since processing them takes minuscule processing time .

Logical expressions can include inequality signs, (< or >) the “is equal” sign (==), ampersands (&) for “and”, a vertical bar (|) for “or” and the squiggly line (~) for “not”. The result for each component and for the overall expression will be either 1 (true) or 0 (false). To be safe, with complicated logical expressions, use parentheses liberally.

As we will see, it is often helpful to use a logical expression on one matrix, then to multiply the resulting matrix of zeros and ones element-wise by another matrix. This sounds far more complex than it is and will seem both reasonable and efficient when used in later chapters.

Matrix Functions That Produce Sub-matrices

There are often times when we are interested in only part of a matrix. For example, the 5'th column of a scenario matrix might contain all the outcomes that could happen in year 5. And the second row might contain all the outcomes for scenario 2. To reference or extract them as a column vector requires only the use of the colon operator, which can be read as "all the entries in the row or column," as the case may be. Here are two simple cases.

```
x =
   1   2   3
   4   5   6

>> y = x(:,2)
y =
   2
   5

>> z = x(2,:)
z =
   4   5   6
```

Here, **y** is the second column of **x**, and **z** is the second row.

It is also possible to create a sub-matrix, requesting an explicit choice of elements. For example:

```
x =
   1   2   3
   4   5   6
   7   8   9

>> y = x( 1:2 , 2:3 )
y =
   2   3
   5   6
```

You can also assign new values to portions of a matrix using this notation on the left side of an assignment statement, as long as the matrix of values on the right side is of the correct size.

Matrices are stored by columns, so you can reference one or more elements using their positions in the list. Thus:

```
x =
    1    2    3
    4    5    6
    7    8    9
>> y = x(6:7)
y =
    8    3
```

While this may seem arcane, it often comes in handy when finding elements using the *find* function and, if desired, changing them. The following example illustrates:

```
x =
    1    2    3
    4    5    6
    7    8    9

>> ii = find((x>=5) & (x<=8))
ii =
    3
    5
    6
    8

>> x(ii) = 99
x =
    1    2    3
    4   99   99
   99   99    9
```

Our last example uses the *diag* function to extract elements along a diagonal in a matrix. As we will see, this is very useful when dealing with mortality tables. The simplest version is illustrated here:

```
x =
    1    2    3
    4    5    6
    7    8    9
>> y = diag(x)
y =
    1
    5
    9
```

The Importance of Matrix Operations

The last part of this chapter may well have seemed tedious and pedantic. But the calculations required to efficiently analyze large scenario matrices tend to be complex. Use of the operators and functions described here (and a few more to be introduced later) can greatly shorten the notation required to document computations and reduce the processing times needed for such computations – in many cases, by orders of magnitude.