Retirement Income Analysis
with scenario matrices

William F. Sharpe

4. Personal States

Retirement Income Strategies

This is a book about *retirement income* – money available to be spent during one's retirement years. Our goal is to provide ways to analyze alternative strategies for providing such income, to find their properties and to help develop new and promising approaches. The methods developed here could be employed by a financial advisor to help an individual or couple choose an overall approach for financing spending in the retirement years. They could be used to identify and reject strategies that seem to be dominated by other approaches for at least certain classes of retirees. And they could be used to create new ways to provide income for future retirees. The possible applications are many and their potential value great.

There is no way that we can deal with all the complex details of any given situation, let alone cover all the possible situations in which retirees may find themselves. Rather we will focus on key choices confronted by many people choosing ways to provide income in the latter part of their lives. Throughout, we will illustrate with a "standard case", discussing but not implementing alternative settings and assumptions along the way. We will focus on couples rather than single retirees in order to cover the most difficult cases. While this limits the possible applications of our software, as we will see, it is possible to approximate a case with a single person by providing him or her with a partner 119 years old. This pretend person will be alive for at most a year, then leave the scene. Inelegant, to be sure, but better than nothing.

This said, it is time to meet the Smiths.

Bob and Sue Smith

The Smiths are our example retirees. They live in the United States. Bob is 67 and has just retired from a position as a University Professor. He will receive monthly payments from the U.S. Social Security System and has a considerable amount in a tax-deferred retirement savings account. Sue is 65 and has just sold her art gallery. She will also receive monthly Social Security payments and has money in her own tax-deferred retirement savings account. Together they have $1,000,000 to finance their expenditures in retirement over and above those covered by the Social Security payments. What should they do? It seems as though every type of financial institution has an answer. Insurance companies are anxious to sell the Smiths annuity policies. Financial advisors believe they can best help Bob and Sue invest their money and spend it at appropriate rates. Mutual Fund Companies have special products designed for people like the Smiths. And so on.

As the baby boomers retire, huge amounts of discretionary investment funds are becoming available for investment by or with the assistance of financial firms and financial professionals. The potential fee income is truly enormous. It is no wonder that the internet, television and publications are replete with ads lauding the superiority of this approach or that over those of competitors. Many are lusting after the Bob and Sue's money and that of others who have recently retired plus the millions who will be doing so in future years.

The Smiths are bewildered. The choices are wildly varied. There are manifold sources of uncertainty. They need help. Our goal is provide some tools that could, in the hands of an unbiased party, be part of a sensible solution.

Personal States

A key aspect of our approach is a focus on alternative *states of the world*. The idea is to identify a set of discrete possible situations for each of a number of key variables. By assumption, at any given time, one and only one of an enumerated set of such states of the world will occur for each variable of interest. The states that concern Bob and Sue's existence we term *personal states*.

We start with such states that are specific to the Smiths. For simplicity we focus on the most basic, with five mutually exclusive and exhaustive states, each indicated by a numeric value:

0. Neither Bob nor Sue is alive
1. Only Bob is alive
2. Only Sue is alive
3. Both Bob and Sue are alive
4. Neither Bob nor Sue is alive for the first time

Throughout, we will deal with the future in terms of discrete years. This will keep the size of scenario matrices relatively reasonable and also conforms with much of current practice. For example, once each year the U.S. Social Security Administration determines a fixed amount to be paid to an individual each month from January through December. Many insurance companies follow a similar approach, adjusting the amounts of monthly annuity payments once each year, with constant payments from January through December. And many popular strategies advocated by Financial Advisors provide a constant monthly payment throughout each calendar year, with the amount determined at or before the beginning of the year.

Our goal is to create a scenario matrix of personal states. Each row will represent a possible future scenario and each column a calendar year. The first column will be “year 1” which starts immediately and extends for 12 months into the future. The second column will be “year 2”. which starts in 12 months and extends for the next 12 months, and so on. In practice these years could start at any date (e.g. October 1st), but to keep things simple, we will assume that each year starts on January 1st. We leave the choice of actual starting dates and other such issues to practical people. The key point is that the beginning of “year 1” is now, and all its attributes are known at the outset.

With these essentials in mind, we can say something about the nature of a scenario (row in our matrix) for the Smiths. First, it must start with a “3”, since both Bob and Sue are alive now. Second, a “3” (both alive) can only be followed by another “3”, a “2” (only Sue alive), a “1” (only Bob alive) or a “4” (neither alive for the first year). A “2” (only Sue alive) can only be followed by another “2” or a “4” (neither alive). Similarly, a “1” can only be followed by a “1” or a “4”. And a “4” can only be followed by a “0” (since more than a year has passed since the first year in which neither Bob nor Sue were alive).

This may seem overly complex. But, as we will see, many sources of retirement income are designed to provide amounts that depend at least in part on the recipients' personal states.

To cover all the possibilities, we need a matrix with enough columns (years) so that every scenario has a “0” or “4” in the final column, to be sure that we cover every possible situation in which Bob and Sue are alive, plus at least one more year to deal with any inheritance. The remainder of this chapter provides methods that can create such a *personal state scenario matrix* for Bob and Sue and, more generally, for others.

Programming Objects, Data Structures and Functions

This book does not aspire to provide a complete programming suite for analyzing retirement income strategies. To do so would require the development of procedures for creating beguiling user interfaces, the inclusion of extensive error-checking, methods for handling special cases, and possibly more concern with execution times. Our goal is instead to design algorithms and programming code that can be used for research and, if desired, employed as major components of a more complete system.

As indicated earlier, our goal is to provide programs that can be executed in Matlab. It now provides for some aspects of *object-oriented programming* – an approach widely used by professional programmers to simplify the implementation of large projects and to reduce the likelihood of errors.

To oversimplify, the idea of object-oriented programming is to construct a series of *objects*, each of which can contain data (*properties*) and procedures (*methods*). For example, a *dog* object might have a *breed* property and a *bark* method. If there were a dog *class* you could then create a new dog object named *fido* as an *instance* of that class. Subsequently you could refer to fido's breed as *fido.breed* and make him bark by executing the method *fido.bark( ).* The key idea is to *encapsulate* the properties and methods of an object together, reducing complexity and the possibility of errors. Fully object-oriented programming languages have these and additional valuable features.

With the community of professional programmers there are many with an almost religious belief in object-oriented programming. On the other hand, there are many who espouse more traditional approaches. We take a middle ground, relying on two constructs (*data structures* and *functions*) that offer some (but not all) of the advantages of true objects while preserving some of the characteristics of traditional approaches.

Data Structures

To keep related attributes of an object together, we employ **data structures**. For example, we could create a dog structure with the following code (with colors provided by the MATLAB editor):

```
dog.breed = 'bichon';
dog.age = 7;
```

We refer to *breed* and *age* as *elements* of a dog structure. These are similar in look and feel to properties of a possible dog object. But there are differences. For example, some properties of a true dog object may restricted to be be “read only”, but any element of a data structure may be changed at will (that is, written as well as read).

Data structures offer convenient ways to keep information together. Elements can be strings, numbers, matrices, etc.. You can make a new data structure by copying the elements of an old one, as in:

```
fido = dog;
```

This will create a new dog structure with the same elements as those of the original one, with the same values (in this example, fido.age will equal 7). But you can change any or all of fido's elements if you wish. And they will all be kept together in the fido structure – a very handy feature indeed.

What about methods? How can we make fido bark? While not as aesthetically pleasing as the use of object methods, we can use traditional *functions* (about which more below). For example, we might create a function called *dog_bark( ),* which won't change any of fido's properties. To make him bark we would simply pass fido's information to a command:

```
dog_bark( fido );
```

But what If we wished to add, delete or change some of fido's elements, say to increase his age after a birthday? Not a problem. We could create a function called *dog_birthday*, then use the command:

```
fido = dog_birthday( fido );
```

This would create a copy of the original fido data structure, make the desired changes to it, then put the revised structure back in the variable named *fido*. Not elegant, to be sure, but reasonably simple, easily understood and not highly error-prone.

The remainder of the book will follow these conventions, using the dot (.) notation to represent an element of a data structure and underscores (_) in the names of functions designed to utilize and possibly change such structures. The goal is to increase clarity and reduce the possibility of errors when the code is utilized.

Functions

To oversimplify somewhat, in Matlab a *function* is a set of code that operates with its own internal information and may produce additional internal information. It can start by copying some external information to its internal variables. And, if desired, it can conclude by copying some of its internal information to external variables. When it is done, all the information created internally is erased.

An example may make this clearer. Consider the following function:

```
function [a,b] = exampleFn( c )
  a = 2*c;
  x = 3;
  b = c/x;
end
```

Now, assume that somewhere in another program you have the following statements:

```
 f = 5;
[d,e] = exampleFn( f );
```

The system will not recognize the reference to *exampleFn(f)* immediately, so it will search through the current directory and any others that you have specified to be on its search *path*, looking for a file named *exampleFn.m;* if it finds one, it will then e*xecute* the function, use the specified input, do its computations, transfer the information from its variables to those in the calling program, then disappear. Let's see what happens in this case.

First the value of **f** in the calling program is copied to the variable **c** inside the function. Next, the commands in the function are executed, creating values for its internal variables **x**, **c**, and **b**. Finally, when the function is finished, it puts the value of its variable **a** in variable **d** in the calling program, puts the value of its variable **b** in variable **e** in the calling program, then throws away all its variables, returns any memory used, and gracefully exits.

This may seem like a lot of effort and memory use. But it has the great advantage of keeping things compartmentalized and keeping errors at bay.

A number of variations are allowed. In our example, the function had one input *argument*, but it could have more, separated by commas, or it could have none (but the parentheses would still be used when defining or *calling* it to make clear that it is a function). Similarly, a function may have one return variable, more than one, or none (in the latter cases, square brackets are optional). The same variable name can be used as an argument when calling a function as in the function definition, but the two will be treated as different variables anyway. This is also the case for any variables returned by the function.

A caveat: be sure to end each function definition with the keyword *end* without a following semicolon, as in this example.

Function Files

As indicated earlier, to be usable directly by a program, a function has to be saved in a file with the function name followed by a .m indicating its file type. Moreover, the file must be in a directory that the system can find – either in the directory being used by the calling program or somewhere on the pre-specified search path. That said, it is possible, and often very desirable, to include two or more functions in a file, with the filename equal to the name of the first function, followed by .m. In such a case, the “main function” (the first in the file) may call other functions in the file, each of which will dutifully return to the main function when through. Note, however, that only the first function in a file may be used by the calling program. The first function in the file is in charge, getting assistance from any other functions as needed, one at a time.

The use of multiple functions in one function file can be very valuable, allowing a complex procedure to be broken into smaller parts, enhancing readability and reducing the probability of error.

An aside: in order to keep things simple we will not attempt to “nest” any functions within other functions.

It is important to remember that, as in Las Vegas, what happens in a function stays in the function, with the exception of information transferred through output variables (on the left of the equal sign in the function definition) to variables in the calling program (on the left side of the equal sign there). This is for your protection.

Function operations may seem overly complex, time-consuming, and prone to indulgent though temporary use of significant amounts of memory. But they offer the promise of power and safety, which usually than compensate for any drawbacks.

The client data structure

With these preliminaries completed, it is time to create a data structure that will store the Smith's information. We could call it *Smith* but that seems too specific. Instead we will use a setting in which Bob and Sue are a *client* of some type of financial advisor (human or electronic). For generality, we refer to person 1 (p1) and person 2 (p2) in programs, but refer to Bob and Sue in many of our discussions.

We begin by creating a function called *client_create( )* that will create a sample client data structure. Here it is:

```
function client = client_create( )
  % create a client data structure with default values
    client.p1Name = 'Bob';
    client.p1Sex =  'M';
    client.p1Age =  67;
    client.p2Name = 'Sue';
    client.p2Sex = 'F';
    client.p2Age =  65;
    client.Year = 2015;
    client.nScenarios = 100000;
   client.budget = 1000000;
  % figure size in pixels: width, height
 %    set to [ ] to use full screen
    client.figureSize = [1500  900];
end
```

Perhaps not surprisingly, initially it contains information for the Smiths.

This should be in a file with the name *client_create.m* so any program can find it when needed. As can be seen, it requires no argument and returns a newly created client data structure. Let's look at it in detail. The first line provides the function information. The second is for readers. In general, any text following a percent sign (%) is considered a comment of no interest to the system. The other lines create elements, then assign default values to them. Most are self-explanatory. Based solely on the letters of their first names, we chose to make Bob person 1 and Sue person 2. We added an element to indicate the year in which we perform the analysis, which will be needed to select appropriate mortality estimates. The next line creates an element to indicate the number of rows (scenarios) in every scenario matrix. In most of our examples we utilize 100,000 scenarios (more on this later in the chapter). The next item is the budget (in dollars) available to finance one or more sources of income. Due to their skills, luck and frugality, Bob and Sue have saved $1 million dollars for this purpose.

The final element is only vaguely related to the client. It controls the size of each of the graphs. Since screens differ in size, it is important to be able to specify a preferred size in pixels. If a screen is narrower than the stated width or shorter than the stated height, the actual dimensions will be used. If this element is set to [ ], ninety percent of the screen will be utilized. And, if the dimensions exceed the screen size, Matlab will adjust the figure size as needed.

We are now ready to build our main program. It will be in a *script file (*any .m file which does not start with a function definition). It can then be executed whenever it's name is referenced from the command line or in a running program. For our example, we'll assume it is named *SmithCase.m*. It starts with the commands:

```
clear all;
close all;
client = client_create( );
```

When executed, this will *clear all* memory (removing any previous values), *close (remove) all* figures currently displayed, then create a client data structure. Since the default information is for the Smiths, this suffices. But for, say the Jones there would be subsequent statements resetting the elements as required. For example:

```
client.p1Name = 'Sam';
```

and so on. But we'll stick with the default information.

Creating Client Personal States

Our goal is to use the information in the client data structure to create a personal state scenario matrix, then add it as an element of the client structure. This will obviously require two sets of estimates of future mortality rates – one for males, the other for females. In this and subsequent analyses ,we will utilize the U.S. Society of Actuaries' RP-2014 mortality rates for 2014 and its MP-2014 mortality improvement rates for 2015 through 2017 and beyond. As shown in the previous chapter, these rates suggest longer lives than those used by the U.S. Social Security Administration, reflecting the evidence that the base group of healthy annuitants receiving benefits from private defined benefit plans had lower mortality rates than the broader group of those receiving benefits from the Social Security program.

We focus on the SOA estimates for three reasons. First, the required data are publicly available, whereas those for Social Security are generally not. Second, the SOA projections have fewer data inputs, since improvement rates are assumed to be constant from 2027 onward. And third, it seems plausible to assume that people with significant discretionary savings at retirement are likely to have lower subsequent mortality rates than the average Social Security recipient. In many years, roughly half of those beginning to receive Social Security have little or no retirement savings. Moreover, the data show that on average they have shorter lives than people in higher wealth categories. Thus it is not unreasonable to assume (as we shall) that Bob and Sue's prospects are better modeled using the SOA mortality estimates than those used by the Social Security Administration. As we will show later in the chapter, cases in which the SOA estimates are too optimistic or too pessimistic might be adequately approximated by adjusting the input age for one or both of the recipients.

To start, we add to the prior statements a single command which calls a function created to do the job:

```
client = client_process(client);
```

This takes the client structure initially created (the *client* on the right), passes its contents to the *client_process* function, runs the function, then replaces the current client structure (the *client* on the left) with the one produced by the function. More simply put, this statement creates the desired personal state scenario matrix, then places it in the client structure.

But where is this function? And how does it accomplish the task?

The answer to the first question is simple. The function is in a file named *client_process.m* in the directory in which the calling program is located or on a path that the processor can follow to find desired functions or script files. To answer the second question we need to see what is in the file. We will discuss this in considerable detail. Those interested only in results may wish to skim over the remainder of the section. Others, with experience in Matlab, may wish to look at the code in the *client_process.m* file available with this ebook. For the rest, here are the details.

The file has, in fact, five functions, each starting with a *function* …. header and ending with *end*. Here is an overview with dots (*…)* standing in for the actual statements in each one.

```
function client = client_process(client);
….
end

function mat = getRP2014Male();
….
end

function mat = getRP2014Female();
….
end

function mat = getMP2014Male();
….
end

function mat = getMP2014Female();
….
end
```

As we know, only the first function can be called from outside this domain. Not surprisingly, in this case the *client_process* function calls each of the other functions, as needed. Their names reveal their jobs. Each returns a matrix with information needed to create a mortality table. The first returns a one-column matrix (vector) with the mortality rates for a male in the year 2014. The second returns a matrix (column vector) with mortality rates for a female in the year 2014. The third returns a matrix with improvement factors for male mortality for the years 2015 through 2027 and beyond, and the fourth returns a matrix with such factors for female mortality. When executed, each provides its data in a matrix called (internally) *mat*.

It is a simple matter to construct these four functions. Within each one, every line has data copied from one of the published SOA documents. Details can be seen in the function listing.

We turn now to the function that does the hard work: *client_process*. We take it in sections. Here is the first part.

```
  % make entire screen white
   ss = get( 0, 'screensize' );
   figwhite = figure;
   set( gcf, 'position', ss );
   set( gcf, 'color', [1 1 1] );

 % compute figure position and add to client
   figsize = client.figureSize;
   if length( figsize ) < 2
     ss = get( 0, 'screenSize' );
    figsize(1) = .9 *ss(3);
    figsize(2) = .9 *ss(4);
  end;
  figw = figsize(1);
  figh = figsize(2);
  ss = get( 0, 'screenSize' );
  x1 = ( ss(3) – figw )/2;
  y1 = ( ss(4) – figh )/2;
  client.figurePosition = [ x1  y1  figw  figh ];
```

The first few statements create a figure that is pure white and covers the entire screen. This way there is no chance that the computer owner's dog or other chosen background will distract from the business at hand. The next create a new data element, *client.figurePosition*, the provides the information needed for subsequent plots – the x and y coordinates for the bottom left corner, the width of the figure and the height.

The next section gets to the main task:

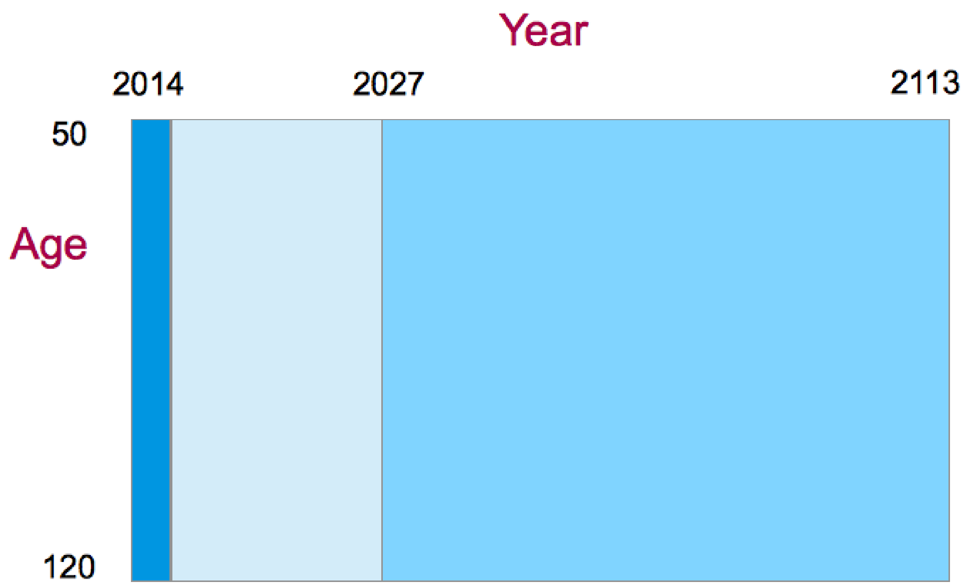
```
% ------------ get vectors of mortality rates for person 1
 % make vectors for rows and columns of mortality tables
   ages = [50:1:120]';       % ages (rows in mortality tables)
   years = [2014:1:2113];  % years (columns in mortality tables)
 % compute mortalities for person 1
   if client.p1Sex == 'M'
      m1 = getRP2014Male();    % get mortalities for 2014
      m2 = getMP2014Male();   % get mortality improvement rates 2015-2027
   else;
      m1 = getRP2014Female();  % get mortalities for 2014
      m2 = getMP2014Female();  % get mortality improvement rates 2015-2027
   end
 % extend mortality improvement rates to 2113 using ultimate rates for 2027
   m2027 = m2(:,13);
   m3 =  m2027*ones(1,86);
 % make mortality table from 2014 through 2113
   m4 = [m1 1-m2 1-m3];   % join 2014 mortality and mortality factors (1 - improvement)
   mortTable = cumprod(m4')'; % multiply 2014 mortality by cumulative factors
 % get vector of mortality rates for p1Age to age 120 beginning in client.year
   rowNumber =  find(ages == client.p1Age); % find row for p1 age
   colNumber =  find(years == client.Year); % fund column for current year
   m5 = mortTable(rowNumber:size(m4,1),colNumber:size(m4,2)); % create new matrix
   mortP1 = diag(m5)';
```

Our initial goal is to create a mortality table with rows for ages from 50 to 120 inclusive, and columns for the years 2014 through 2113. These commands do so for Males. First we create a column vector with the ages (note that we create it first, then transpose using the ' operator to make it a column vector), Next, we create a row vector with the years. Then we use the appropriate functions for the sex of person 1 to get mortalities and mortality improvement factors, placing the mortalities in column vector *m1* and the improvement factors in matrix *m2*.

Next we extend the mortality improvement factor table by creating additional columns that are replicas of the column for 2027, conforming to the assumption that the rates for 2027 are ultimate rates that will be repeated in all subsequent years. Then we create a matrix (*m4*) with the 2014 mortality rates, followed by the complements of all the improvement factors for the years from 2015 through 2113 (since the SOA tells us that an improvement factor of *x* (e.g. 0.01) means that the mortality in a given year is *1-x* (e.g. 0.99) times that in the prior year).

The next step shows again the power of matrix operations. We simply multiply each column of the *m4* matrix by the cumulative product of the predecessor columns, providing a table of mortality rates for each age and year. (first using the transpose, then transposing the result to make cumulative products for each row). This works because the initial mortality rate is multiplied by the first improvement ratio to provide the second mortality rate, then this is multiplied by the next improvement factor to provide the third mortality rate, and so on. Simple, safe and fast. The result is the matrix named *mortTable*.

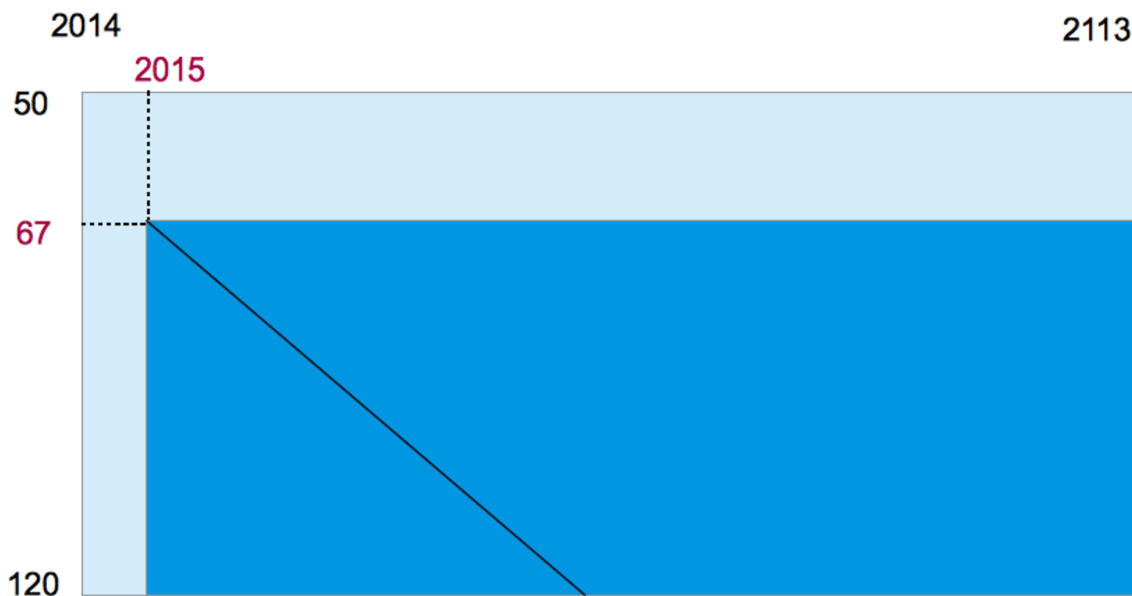
The figure below represents the components of *mortTable*. The first column contains the mortality rates for 2014 from the RP-2014 report. The next 13 contain projected mortalities for 2015 through 2027 based on the mortality improvement factors for each year in the MP-2014 report. The final columns contain projected mortalities for years after 2027, computed usingthe MP-2014 assumption that the 2027 improvement factors are appropriate for all subsequent years.



The next task is to extract from this table Bob's mortality rates from 2015 until he reaches (or would have reached) the ripe old age of 120. Here are the statements:

```
% get vector of mortality rates for p1Age to age 120 beginning in client.year
   rowNumber =  find(ages == client.p1Age); % find row for p1 age
   colNumber =  find(years == client.Year); % fund column for current year
   m5 = mortTable(rowNumber:size(m4,1),colNumber:size(m4,2)); % create new matrix
   mortP1 = diag(m5)';
```

The diagram below illustrates what they do.



We take the full mortality matrix, then create a sub-matrix starting with the row for Bob's age (67) and the column for the current year (2015). Next we take the values on its principal diagonal, representing the mortality rates for Bob's cohort, putting the results in the vector *mortP1*. The values end when Bob's age is 120. All this in four statements!

The next portion of the function includes statements that do the same operations for Sue, creating a vector of her mortality rates, not surprisingly, called *mortP2*.

We are almost ready to create the scenario matrix. But a bit of housekeeping is required first. We need to extend the shorter of the two mortality vectors so they will be the same length. These statements do the trick (don't worry about special cases, a vector of ones of zero length is an empty vector, as any mathematician would expect).

```
% ----- extend shorter of mortality vectors to length of longer
   ncols =  max(length(mortP1),length(mortP2)); % number of columns longer vector
   mortP1 = [mortP1 ones(1,ncols-length(mortP1))];
   mortP2 = [mortP2 ones(1,ncols-length(mortP2))];
```

The next adjustment arises from the fact that we intend to specify the status of our clients at the *beginning* of each year, and at the beginning of year 1 both Bob and Sue are alive and well. We can handle this by starting the mortality vectors with a zero rate so that neither of them dies prematurely. This will require a scenario matrix with one additional column. Here are the statements.

```
% ---- add zero mortality for year 1
  mortP1 = [0 mortP1];
  mortP2 = [0 mortP2];
  ncols = ncols+1;
```

For possible later use we also add these vectors to the client data structure:

```
% ---- add mortalities to client data structure
  client.mortP1 = mortP1;
  client.mortP2 = mortP2;
```

We are finally ready to make the first scenario matrix. We want a matrix in which an entry of "1" indicates that Bob is alive in that scenario and year and an entry of 0 indicates that he is dead (or a euphemism of your choice). Very few statements are needed. Here they are.

```
% make personal state matrix
  nrows = client.nScenarios;  % number of rows in scenario matrices
  % person 1
    probs = ones(nrows,1)*mortP1; % probabilities of dying
    randnums = rand(nrows,ncols); % random numbers
    statesP1 = double(randnums >= probs);    % 1 if alive, 0 if dead
    statesP1 = cumprod(statesP1')';     % survivals, 1 if alive, 0 if dead
```

First, we set the number of rows for our scenario matrix to the number of scenarios requested by the client. Next we make matrix *probs* which has Bob's mortality for each year in every row for the appropriate column. This represents the probability that Bob will die in that year, if he is in fact alive. Next we create a matrix full of random numbers, each drawn from a uniform distribution of values between 0 and 1. This is easily done since Matlab has built-in random number generators. The one we need is called, simply, *rand( )*.

The next statement performs a very useful trick. It compares the random number in each cell with the mortality probability in the cell and creates a new variable to place in a matrix called *statesP1* which will equal 1 if the random number is equal or greater than the mortality probability and 0 otherwise. We take advantage of the fact that Matlab represents the logical value *true* with a *1* and the logical value *false* with a 0. In most cases, these can be considered numeric values and used as such. However the *cumprod* function deals only with numeric values, so we convert the numbers produced in the logical test to numeric (double precision) values using the *double( )* function, putting the results in a new matrix called *statesP1*.

We have but one statement left in this section. Recall that each 0 entry in the initial version indicates that Bob will die if he is alive and each 1 entry indicates that he will live if he was previously alive. But of course if Bob were already dead in a year, he will remain so. The last statement takes care of this unfortunate aspect of real life. We simply take the cumulative product of all the entries in each row (scenario). This insures that once dead (0), Bob will remain so. The result is a new version of *statesP1* with 1's for each year in a scenario in which Bob is alive and 0's for all remaining years. We have the desired scenario matrix for Bob.

The next task is to do the same thing for Sue. The procedure is exactly the same but we put the results in a matrix called, not surprisingly, *statesP2*. There is, however, one additional step. We wish this matrix to have a 2 in every case in which Sue is alive and 0 when she is dead. Simple, we add:

```
statesP2 =  2*statesP2;  % code as 2 for person 2
```

The next step is to create a matrix with codes for both Bob and Sue, containing a 3 for states in which they are both alive, a 2 for states in which only Sue is alive, a 1 for states in which only Bob is alive and 0 for states in which neither is alive. It is much easier to write the statement than to explain its goal. It is:

```
% add person 1 and person 2
  states = statesP1 + statesP2;
```

We now have a scenario matrix *states*, which is almost ready to put in the client structure. But there is one last task. For each scenario, we want to have an entry of 4 for the year following the death of the last survivor. This can be easily done, although the statements may seem a bit formidable. Here they are:

```
% add estate (4) whenever 0 preceded by 1,2, or 3
   estates =  (states(:,2:ncols) == 0) & (states(:,1:ncols-1) > 0);
   estates = [zeros(nrows,1) estates];
   states = states + 4*estates;
```

The first statement uses two matrices, each created from the current one. The first matrix uses all but the first column of the *states* matrix, while the second matrix uses all but the last column. We can then look for situations in which there is a zero in the second matrix and a number greater than zero in the first. In each such case, the last survivor has just died. We take the resulting matrix, add a column of zeros to make it compatible with our initial states matrix, multiply by 4 to give our desired code, add it it to the original states matrix and we have what we want. All that is left is to put it in the client structure, then return the new structure to the calling program. The final statement is thus:

```
% put in client scenario matrix
   client.pStatesM = states;
```

And the function's major task is done. Subsequent statements provide values for the two persons' life expectancies. More on that later.

While we do not need to take advantage of the fact, it may be helpful to consider the way in which the codes we use for personal states would be represented as binary numbers (using only 0's and 1's, in which each position represents a value twice as great as the one to its right). In this form, the code for *Bob only alive* would be 1, the code for *Sue only alive* would be 10, that for *both alive* would be 11, the code for the year after the last died would be 100 and the code for all other states would be 0. Handily, each position represents one recipient (from right to left: Bob, Sue, and the estate). In a more general approach, one could include additional states. For example, a binary "1" to the right of the point might indicate that Bob requires long-term care, a "1" two places to the right might indicate that Sue requires such care, and so on. We shall not attempt this, however, using only five states: 0,1,2,3 and 4 in traditional (decimal) numbers – equivalent to 0, 1, 10, 11 and 100 in binary.

This aside completed, let's return to our function. It obviously does a great many calculations and produces a very large matrix, with well over five million numbers for a case with 100,000 scenarios. One might imagine that it would take a great deal of computer processing time. Surprise! On the author's Macbook computer the entire task takes less than one second. Such is the power of a language designed and implemented to perform matrix operations efficiently.

Survival Rates

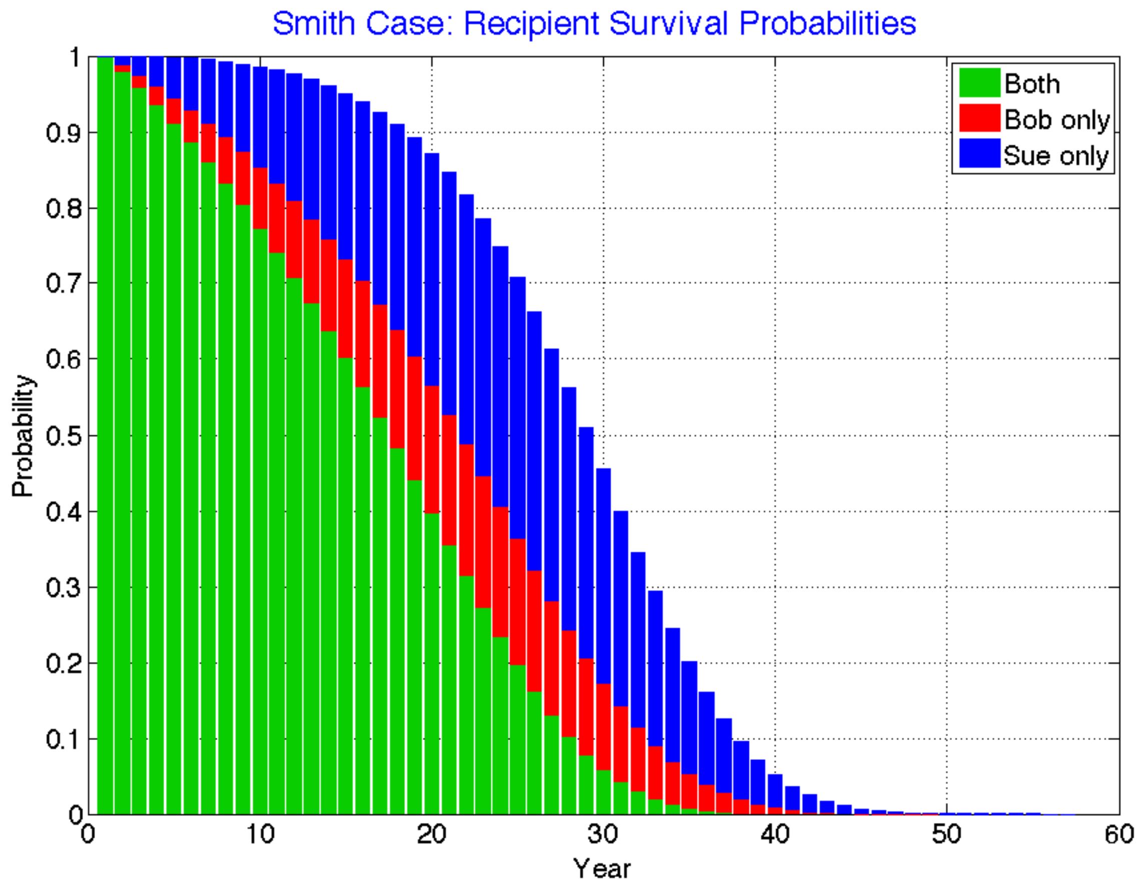
Before Bob and Sue consider alternative retirement income strategies, they should seriously study the probabilities of the alternative personal states in each future year. How likely is it that they both will be alive at the beginning of the second year? The third? And so on. How likely is it that only Bob will be alive in each future year? Only Sue? We can show the answers in a *Survival Probability Graph.*

It is a simple matter to get and plot the needed information. Here are the key statements:

```
probSurvive1only =  mean(client.pStatesM == 1);
probSurvive2only =  mean(client.pStatesM == 2);
probSurviveBoth =   mean(client.pStatesM == 3);
probSurviveAll = [probSurviveBoth ; probSurvive1only; probSurvive2only]';
```

Consider the first statement. For each year (column) we seek to find the proportion of scenarios in which only client 1 is alive (that is, the personal state equals 1). To do so we create a matrix in which there is a true (1) entry if the condition is met and a false (0) entry otherwise. If desired, we could then apply the *sum( )* function, which would provide the number of scenarios for each year, then divide these totals by the total number of scenarios (100,000) to get the desired proportions. But we can get the desired result in one statement by simply computing the mean values. The result is *probSurvive1only*, a row vector of the probabilities of that only Bob will be alive.

The next two statements create row vectors with the probabilities for the other two states of interest (only Sue alive and both Bob and Sue alive). Next we create a matrix with the row vectors stacked on top of one another, then transpose it to accommodate the conventions of the plotting routine. Then we can add statements to create axis labels, a title, a grid, a legend, choose desired colors, etc. (details will be shown in Chapter 11). The next figure shows the graph with these added features:



Several aspects of these survival rates warrant discussion. First, the probabilities that both will be alive diminish year by year, reaching almost zero 38 years hence. The probabilities that only Bob will be alive are initially relatively small, growing for the first twenty five years, then diminishing. The chances that only Sue will be alive are also initially relatively small, but grow to be considerably larger before diminishing. The reasons are not mysterious. Sue is younger and female, and both factors contribute to make her mortality rates smaller than those of the older and male Bob.

Notice that we use green to represent states in which both are alive, red for states in which person 1 (here, Bob) only is alive and and blue for states in which person 2 (here, Sue) only is alive. We adopt this convention in other graphs as well.

The survival rate graph is not of simply intellectual interest. A strong case can be made that every client should study his, her or their survival graph and seriously consider its implications. Unfortunately, many people do not wish to do so. It is not pleasant to be confronted with information about the prospects for one's own death and likely those for a partner as well. And it is hard to confront the prospect that either person could end up alone for possibly many years. As we will see, many people who have already thought about their mortality have done so in terms of a single number, such as the expected number of future years of life. This is not surprising -- for many people, probability distributions are neither familiar nor intuitive. Moreover, in this case serious emotions are involved. The possibility of a short life is depressing on its own grounds, while the possibility of a very long life is depressing on financial grounds. Discussions about mortality (longevity) are likely to be fraught.

No wonder many retirees simply do not choose to try to internalize the information in a survival rate graph. But it is imperative that they do so. Only then can there be intelligent conversations about the desirabilities of alternative sources of retirement income.

A financial advisor's first task should be to help his or her clients study their survival rate graph. The experience could well be that described by one of the author's friends, who commented after he and his partner studied their graph that “It generated a number of discussions between us”. Painful, perhaps. Difficult, certainly. But essential.

Sampling Error

We have drawn the survival probabilities from our 100,000 simulated scenarios. But we could as easily have computed them directly from the mortality rates of our two clients. In a sense, these are the "true" survival rates (if the mortality rates are themselves correct). Using standard statistical terminology, we can consider these the *population statistics*, while the rates based on our simulations are *sample statistics* drawn from the overall population. The former will be the same from case to case, but the latter will vary. The difference can be considered *sampling error*. The larger the sample (number of scenarios), the smaller should be the error.

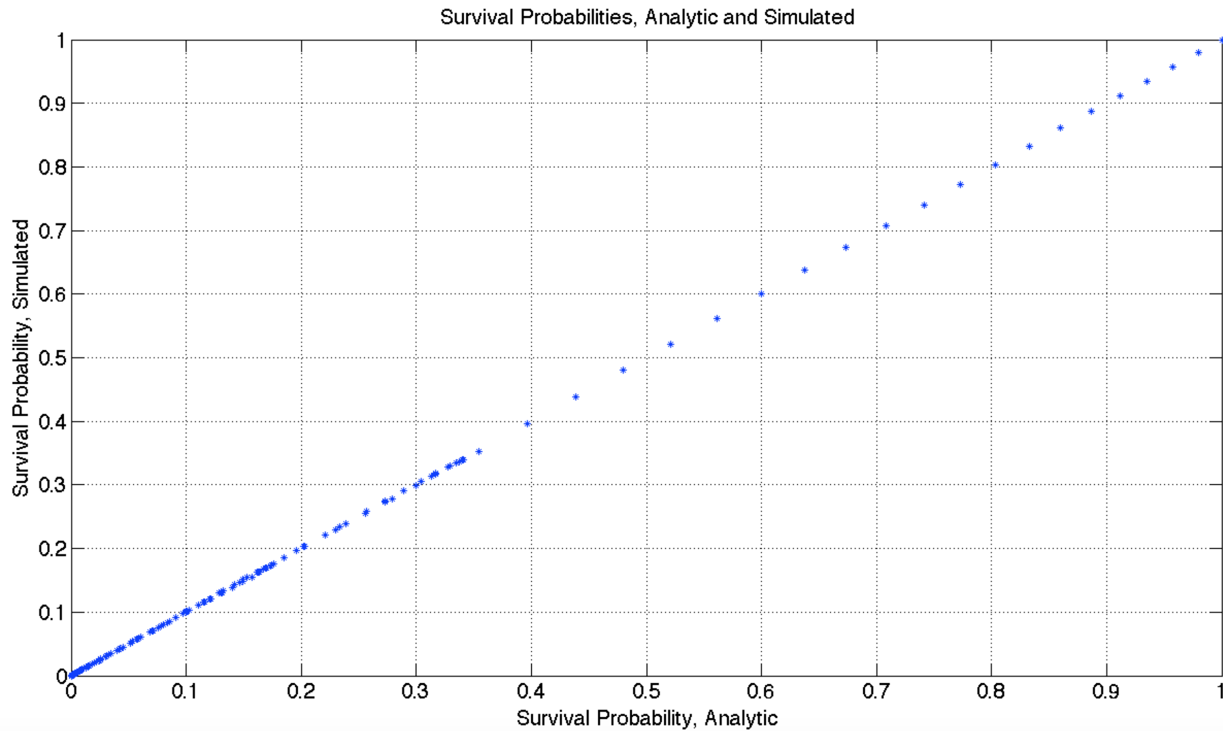
This raises an obvious question: are 100,000 scenarios sufficient to reasonably represent the range of survival probabilities? To provide an answer, we start by computing the population survival rates. This is straightforward: here are the statements:

```
surv1 = cumprod(1-client.mortP1);
surv2 = cumprod(1-client.mortP2);
survboth = surv1.*surv2;
surv1only = surv1.*(1-surv2);
surv2only = surv2.*(1-surv1);
```

A survival probability is equal to 1 minus the mortality probability (you survive if you don't die). The probability that you survive for 2 years is the probability that you survive in the first year times the probability that you survive in the second year. And so on. We use the *cumprod( )* function to determine the vectors of survival probabilities for each person.

Now to the joint probabilities. Assuming (perhaps unrealistically) that Bob's survival probabilities are the same whether or not Sue is alive and that Sue's probabilities are also unaffected by whether or not Bob is alive, the probability that both Bob and Sue will have survived to a given year will equal the probability that Bob has survived times the probability that Sue has as well. Multiplying these for each pair of values (using the element-wise operator .*) gives the survival probabilities for both. Next we compute the probabilities that only 1 (Bob) has survived for each year by multiplying the probability that he has survived times the probability that Sue has not (that is, 1 – the probability that she has survived). A similar calculation gives the probabilities that only Sue has survived.

We can now compare each probability for the population with the corresponding probability in the sample (scenario matrix). The result is shown in the figure below.



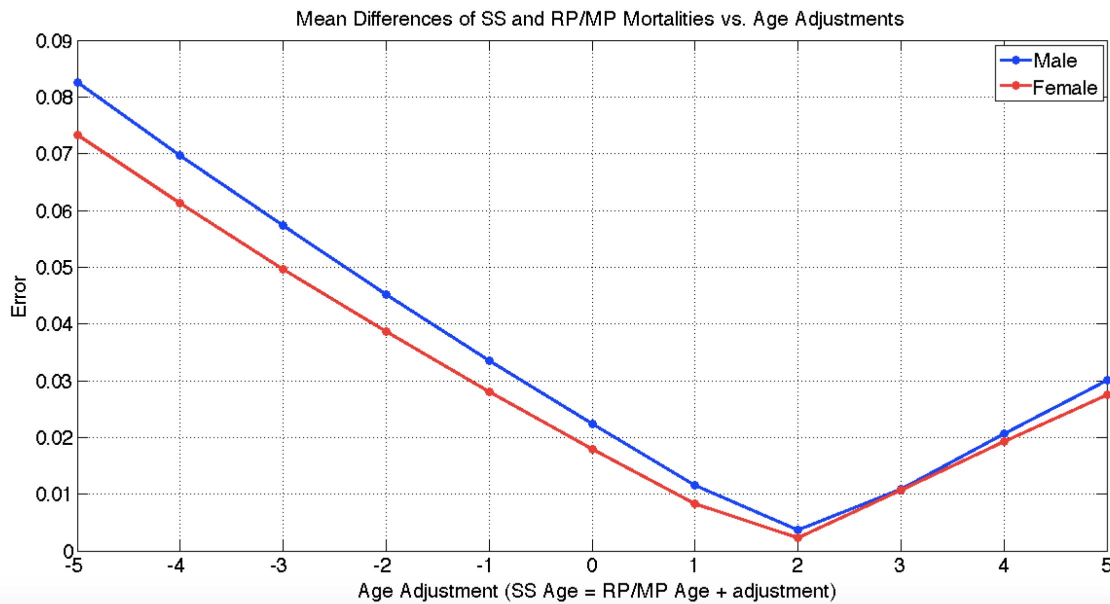
The results are very encouraging. Each of the frequencies in our scenario matrix is very close to or equal to the "true" probability. Indeed, when a regression was performed to find the line of best fit, the intercept was 0.0001 (very close to the perfect value of 0.000) and the slope was 0.9999 (very close to the perfect value of 1.0000). Moreover, the R-squared measure of the fit is 0.999993 (remarkably close to the perfect value of 1.0000). To be sure, another run of the program, using different random numbers, would produce different scenarios, with a different set of errors. But it appears that for participant survival rates, 100,000 scenarios suffice to produce highly representative results.

Age Adjustments

We turn now to questions concerning the appropriateness of the RP-2014/MP-2014 mortality rates for particular clients. As indicated in the previous chapter, these rates are based on experienced and projected mortalities of a population of people who choose to annuitize private corporate defined benefit pensions and are healthy at the time they begin receiving payments. But mortality rates for other groups may be different. We saw in the previous chapter that mortality rates for recipients of Social Security payments are in fact greater. What if Bob and Sue are more like the typical Social Security recipient than the typical person in the pool used to construct the RP-2014/MP-2014 tables? Is there still a way to use the programs that we have developed?

Perhaps. We could employ our mortality tables but better represent the likely experience of a particular person by entering a different current age as an input. For example, although Sue is 65 we might enter her age as 67 to obtain more appropriate mortality rates. In other words, we would use an *age adjustment* of +2, then proceed as usual.

But how to select the most appropriate age adjustment? Consider the differences between Social Security and RP-2014/MP-2014 mortalities. The figure below shows a measure of the error associated with using the RP/MP table with an age adjustment when the SS table is the correct one. For example, an adjustment of +2 indicates that we would input a 65-year old's age as 67, assuming that a 65-year old Social Security recipients will experience mortality rates most like those of 67-year old healthy private pension annuitants. The error measure on the vertical axis takes into account the errors at every age from 50 to 100. More precisely, the measure is is the square root of the mean of the squared differences between the corresponding mortality rates. The figure indicates that in this case, the best adjustment (among possible integer values) is to increase the input age by 2 for both males and females. Thus we might input an age 67 for the 65-year old Sue and an age of 69 for the 67-year old Bob.



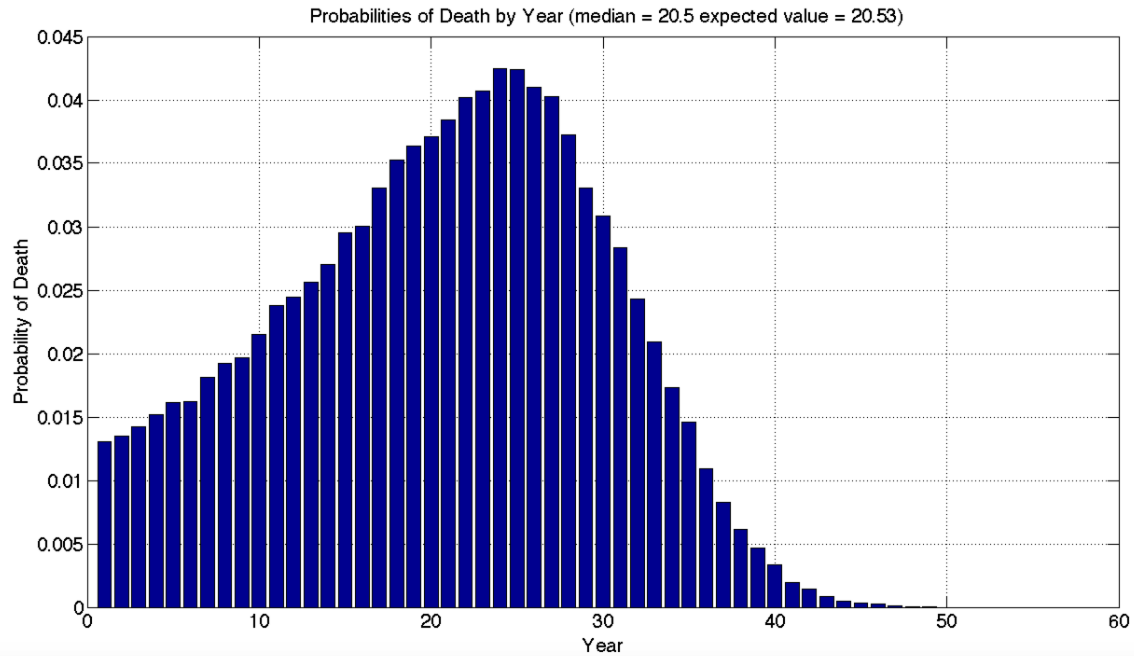
Some would conclude that a typical 65-year old Social Security recipient could be considered to have a "biological age" of 67 when the RP-2014/MP-2014 tables are used to project mortality. But of course such a value is specific to the tables used. In this example, if we were using Social Security tables the age adjustments would equal 0 and the "biological age" would equal the chronological age. The term is at best not well defined and at worst can be misleading. Better to simply focus on finding an appropriate age adjustment that will provide reasonable mortality estimates, given the tables employed.

Life Expectancy

Our comparison of RP/MP and Social Security mortality tables was intended to be illustrative rather than practical. If we have the correct mortality table for a person, why not use it, rather than an approximation obtained by applying an age adjustment to an incorrect table? But sometimes we have only one or two statistics from a presumably more relevant mortality table,but not the full set of mortality probabilities. By far the one most commonly used is an estimate of a person's *life expectancy*.

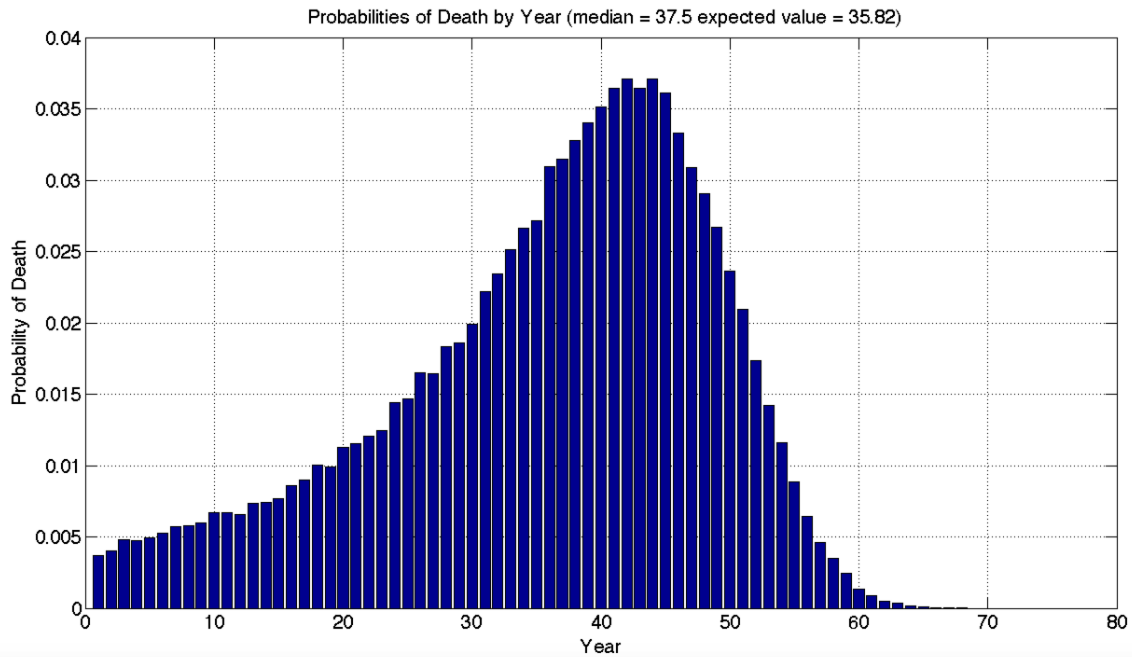
Many assume that life expectancy (x) is computed so that it fits the statement "There is a 50% chance that you will live to age x". Not so. This is the *median,* a value above which fall half the remaining observations and below which fall the other half (or, if the number of observations is even, the average of the two observations in the middle). The expected value for a mortality distribution is instead computed by multiplying each future year by the probability of being alive in that year, then taking the sum of the products. For a set of realized values, this is typically called the *mean;* when probabilities of future values are used, the result of such a computation is more commonly called the *expected value*.

Let's illustrate with Bob's prospects for living after his present age of 67. The following figure shows the probabilities that he will die in each future year.

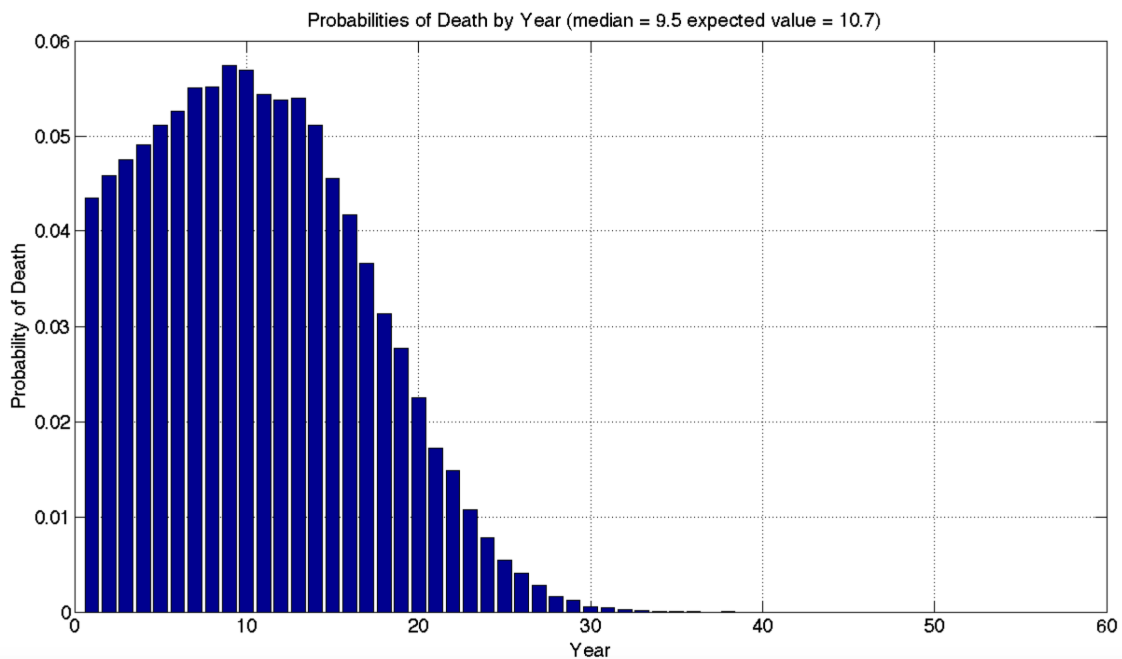


As the caption shows, in Bob's case the median and expected values are almost identical.

But what if Bob were 50? The graph below shows the probabilities of death in that case. Due to the extreme skew of the distribution, with a long "left tail", the median is greater than the expected value (mean).

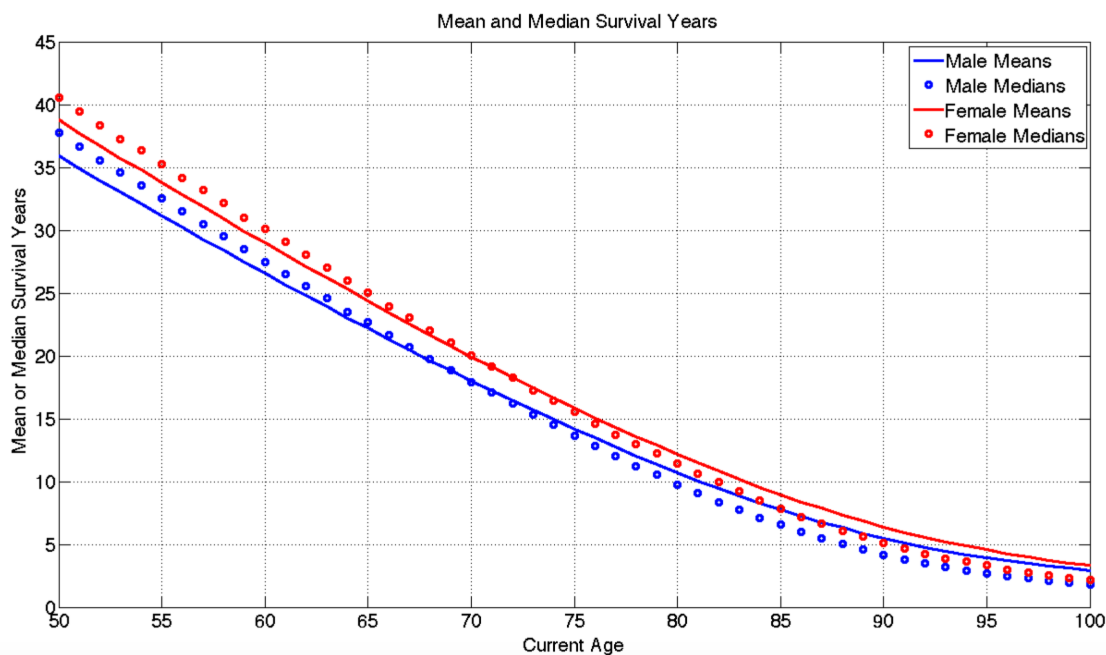


If Bob were 80 the result would be just the opposite, as the following graph shows:



These graphs show the influence of two opposing forces. The probability of death in year t depends on the probability of surviving until year t (the survival probability) and the probability of dying in that year if you are alive (the mortality probability). Over time, the former decreases and the latter increases in a way that causes their product to increase, then decrease, and the strength of these forces are different for younger people than for older folks.

The following figure shows the overall results for males and females of different ages. For 70-year olds the median and mean number of survival years are similar, but for those considerably younger or older, they can differ by as much as two years.



Clearly, it is better not to assume that you have a 50% chance of living up to the age that someone has computed as your "life expectancy". More importantly, it is dangerous to focus on only one possible length of a possible future life. It is the premise of this book that one should think about the future as a large set of possible scenarios, not one. Far better to focus on the entire set of possible lengths of life summarized in our survival rate graph.

But why then does the *client_process* function compute life expectancies? For only one reason: to allow comparisons with results obtained using other procedures so that informed age adjustments can be made if and when they seem advisable.

We turn finally to the murky area of life expectancy questionnaires, many of which can be found on that fount of information and misinformation – the Web.

Life Expectancy Questionnaires

Many web sites will happily provide estimates of a person's life expectancy (sometimes called *lifespan*) based on answers to a number of questions concerning personal habits, medical history, and other purportedly relevant factors. The sites vary widely in information requested, the underlying mortality estimates utilized, degree of objectivity and the extent of documentation of procedures. Many attempt to sell the user some product or service: medications, books, self-help manuals, insurance policies, exercise equipment,... whatever. Underlying methods are rarely documented in detail. Moreover, results can vary dramatically from site to site. One cannot help but wonder about the quality of the underlying research. That said, some of the information might possibly be germane for estimating a person's mortality. At the very least, one may find some of the sites amusing.

Of course, life expectancy depends on the mortality table utilized, and there appears to be considerable variation among the underlying mortality tables used on such sites. In most cases, no references are given. And at least one case uses a period table, with no mortality improvements at all (that is, it employs a vertical column in a mortality table, rather than a diagonal). Not surprisingly, the resulting life expectancies were quite short.

To provide a more reputable example, we obtained some results for Sue, a 65-year old woman of average height and weight, in early 2015 from the "Lifespan Calculator" provided by Northwestern Mutual Life Insurance Company, a large and venerable company in the United States. After asking for the subject's age, gender, height and weight, the site provides a series of multiple choice questions. After the user makes a choice for each question the site shows a projected life expectancy based on the answers provided to the qyestions. To see the effects of different responses, one can try alternative answers to a question, then observe the resulting lifespan. A little experimentation showed that the range of such estimates could be quite large. The following table shows each of the questions along with the best (highest life expectancy) and worst (lowest life expectancy) allowable answer for each, as well as the range of differences in life expectancies obtained by varying the answers to the question, with moderate answers given for the remaining questions.

Family Cardiovascular History (range = 4 years)
best: Family member lived to age 70 with no cardiovascular problems before age 55
worst: 2 or more family members with cardiovascular problems before age 55

Blood Pressure (range= 6 years)
best: Blood pressure checked regularly with normal reading
worst: High blood pressure, not under control

Stress (range = 2 years)
best: Stress is a positive influence
worst: Stress often overwhelms me

Exercise (range = 6 years)
best: Daily vigorous exercise
worst: Not active

Diet (range = 5 years)
best: Eat more than 5 portions of fruits and vegetables
worst: Eat fast or processed foods regularly, and minimal vegetables

Seatbelt (range = 4 years)
best: Always buckle up
worst: Do not always buckle up

Driving (range = 13 years)
best: No accidents/violations in past 3 years
worst: More than one DWI conviction in past 5 years

Drinking (range= 7 years)
best: Don't drink or never drink more than 2 drinks a day
worst: 5 or more drinks at one time, more than once a month

Smoking (range = 10 years)
best: Never smoked
worst: Smoke 2 or more packs per day

Drugs (range = 9 years)
best: Never use drugs for “recreation”
worst: Use drugs for “recreation”

Doctor Visits (range= 2 years)
best: I regularly schedule check-ups with a physician
worst: I never visit a doctor

One can't help but be struck by the large ranges in predicted life expectancies associated with different answers to some of these questions. At the very least, taking the questionnaire might lead to changes in behavior. Perhaps that fifth drink once a month isn't worth it. Your friends that suffer through vigorous exercise every day may just be on to something. And certainly you should never, never fail to buckle that seatbelt.

After investigating the ranges of possible life expectancies associated with different answers to the Northwestern Mutual lifespan calculator, we went through the questionnaire providing the moderate answers that described Sue's situation. Her resulting life expectancy estimate was age 89, that is 24 future years, given her current age of 65. This compared well with the value computed by our software: *client.p2LE* was 24.4 years. Thus it makes sense to continue to enter Sue's age as 65. Of course, if there were a substantial disparity, and we were convinced that the questionnaire was right, we could have tried entering a different age, running *client_makePStatesM(client)*, examining the value of *client.p2LE,* then repeating the process with different ages until the associated life expectancy conformed with that obtained from the questionnaire.

Well and good, but how can one evaluate the accuracy of any web site that purports to make predictions of life expectancy based on answers to questions concerning personal habits and history? It is, at best, difficult. Different sites consider different factors and can provide radically different forecasts. Moreover, some provide estimates with ludicrous degrees of precision. For example the *Blue Zone Vitality Compass,* which claimed to be "the most accurate life estimator available" not only provided an overall life expectancy but also indicates the number of additional days that can be gained by changing one's behavior. In Sue's case, the three most effective changes in early 2015 were:

Eat Whole Grains – Gain 304 days

Have a Little Faith (religion) – Gain 195 days

Improve Your Attitude – Gain 170 Days

While these online aids could be entertaining, one might be justified in leaving well enough alone and simply accepting the mortality estimates computed from the RP-2014/MP-2014 tables.

The *client.incomesM* matrix

Whether one chooses to enter a client's actual age or an adjusted version, our programs will provide a complete matrix of possible personal states for many scenarios and many years. We assume (a) that only one of these scenarios will actually occur, (b) that at present we do not know which one it will be, and (c) that each has an equal probability of occurring.

The *client.pStatesM* matrix is our first scenario matrix. As we will see, its size will determine the size of every other scenario matrix, so that each will have the same number of scenarios (rows) and years (columns).

The goal of any retirement income strategy is to produce incomes in appropriate scenarios, years and personal states. All the needed details can be provided in a single matrix showing the real income to be obtained in each possible scenario and year. To prepare for future analyses we create a matrix with zero in every cell, using the command:

```
% create empty client incomes matrix
    client.incomesM = zeros(size(client.pStatesM));
```

Importantly, any retirement income strategy should be personal, hence the personal states matrix will influence the contents of this and some other scenario matrices. First, however, we need to generate matrices of values for external variables such as security returns, inflation and present values. These are the subjects of the next chapter.