Retirement Income Analysis
with scenario matrices

William F. Sharpe

11. Analysis

Overview

The previous chapter provided methods for producing income and fee scenario matrices for different types of fixed annuities, then adding these to any previous values for such matrices in a client data structure. In an important sense, these matrices are the focus of this book. Our view is that a retirement income strategy or combination of such strategies should be viewed as producing such matrices, then evaluated based on the characteristics of the matrices. We prefer the term *scenario matrices* to *Monte Carlo analysis,* since the former encourages a focus on the complex multi-period, multi-state probability distribution reflected in these large matrices.

Unfortunately, few if any human beings can evaluate two or more matrices for a strategy, each with millions of entries, let alone compare two or more alternative strategies, each with several scenario matrices. We need analytic tools to summarize and represent key properties of such matrices. The focus of this chapter is on a way to organize a set of such analyses. Subsequent chapters will introduce specific analytic approaches and illustrate their use with different types of retirement income strategies.

The Analysis Data Structure

The vehicle we will use to organize a set of analytic methods will be (not surprisingly) a data structure called *analysis*. It contains information concerning desired output formats and variables that can be set to indicate which types of analyses are to be made in any given case. As with our other data structures, it is possible to create a generic version, make changes in its elements as needed, save the structure under an appropriate name, then subsequently load a copy when desired.

The elements of an analysis data structure indicate the analyses to be performed as well as outputs to be produced in each case. As usual, we divide the process into two parts – creating the structure, then processing it.

Creating an Analysis Data Structure

To create an analysis data structure *de novo* we use a command of the form:

analysis = analysis_create( );

We will build a basic *analysis_create* function in this chapter, then add elements to it in subsequent chapters. Eventually, we will have a number of analytic tools, any one of which can be applied by setting values for one or more elements in the analysis data structure, then processing the structure. Many of these methods depend on the statistical and economic models and assumptions described in previous chapters, in particular those associated with the pricing kernel and market portfolio probability distributions. We will devote a great deal of attention to prices and present values (as one would expect in a model developed by an economist). Of course, results will depend on underlying assumptions, both those incorporated in the programs and those set by changing data structure elements. As usual, one hopes that the term "garbage in, garbage out" will not apply.

Here is an initial version of the *analysis_create( )* function, including an element to indicate whether or not to depict survival rates.

```
function analysis = analysis_create(  );
  % create an analysis data structure
  % case name
     analysis.caseName = 'Smith Case';
 % animation first and last delay times
     analysis.animationDelays = [ 1  0.5 ];
% animation shadow shade of original (0 to 1)
    analysis.animationShadowShade = 0.2;
% delay time between figures (0 for beep and keypress) in seconds
     analysis.figureDelay = 0;
 % stack figures or replace each one with the next
     analysis.stackFigures = 'n';
% close figures when done
     analysis.figuresCloseWhenDone = 'y';

 % compute and plot survival probabilities -- y (yes) or n (no)
     analysis.plotSurvivalProbabilities = 'y';

end
```

The *caseName* should be set to describe the particular case being analyzed, including the types of strategies used to create incomes.

The next statements provide parameters for graphs that use *animation* to show multiple relationships on a single figure. For each relationship: (a) one set of data is plotted in dark shades of selected colors, then (b) after a delay, that plot is redrawn in a lighter shades, after which (c) there is a timed delay. This process is repeated until all the relationships have been plotted.

The length of the delays is given by the two parameters in the analysis.animationDelays element. The first parameter indicates the length of the first delay (here, one second) while the second indicates the length of the last delay (here, half a second). As the animation proceeds, delays will change by equal amounts to move from the first to the last delay time.

The next element indicates the proportion of the initial shade of each relationship to be used after it has been succeeded by another. This can be set from 0 (in which case the original information will disappear completely) to 1 (in which case there will be no change from the original plot).

The *figureDelay* element indicates whether there should be a fixed time between showing a figure and the next one. If the value is positive, it indicates the number of seconds between figures. If it is zero, the processing program will sound a *beep* after each figure, then wait for the user to press a key such as the space bar before continuing.

The next two elements indicate (a) whether figures should be stacked, one on top of another, or each should replace its predecessor and (b) whether or not figures should be closed after they have all been shown. If many figures are to be shown, it is preferable that they not be stacked, since this can require substantial memory and may overtax the Matlab processor.

The last element, *plotSurvivalProbabilities*, indicates whether or not the survival probability graph (which we saw in chapter 4) should be produced. No additional information is needed to create the graph when the analysis data structure is processed.

Subsequent chapters will introduce additional elements to be included in the analysis data structure. Such elements can determine whether particular analyses are to be performed and, if so, provide needed values. Since the survival probabilities figure requires no such parameters, a single element indicating whether or not to produce it suffices.

Processing an Analysis

As with other procedures, we divide the analysis task into two operations – creating a data structure, then processing it. We illustrate the latter with a function that can produce the recipient survival graph described in Chapter 4. Subsequent chapters will add statements to both the data structure and the function for processing it.

Here is an initial version of the *analysis_process* function.

```
function analysis_process( analysis, client, market )
  % process an analysis data structure to produce analysis output

  % initialize
     analysis = initialize( analysis, client  );

  % Plot survival probabilities
     if analysis.plotSurvivalProbabilities == 'y'
       % create figure
          analysis = createFigure( analysis, client );
       % call external function analPlotSurvivalProbabilities
          analPlotSurvivalProbabilities( analysis,  client,  market );
       % process figure
          analysis = processFigure( analysis );
     end;

  % finish
    finish( analysis );

end % function analysis_process( analysis,  client,  market )
```

Note that the function uses three data structures (*analysis*, *client* and *market*) but does not return any outputs. Since it is crucial that arguments for functions be the same order in the function and when called. As we will see, the function does make changes to its internal version of the analysis data structure, but since this version is not returned to the calling script, the original structure is unchanged after the analyses are performed.

The function begins by calling another function, *initialize*, which is included in the same file. It then calls three other functions, one that is stored in another file, and two included in the same file as the *analysis_process* function. This works because whenever Matlab encounters a call to a function, it searches for the function first in the current file; if it is not found, Matlab searches in the current directory or other directories on the its current path.

The central section of the *analysis_process* function contains instructions for creating plots, if and when desired. Here we show only the section for the survival probabilities graph. If the *analysis.plotSurvivalProbabilities* element is *'y'*, the figure will be created. There are three steps. First a figure is created using the *createFigure* function contained in the *analysis_process* file. Then function named *analPlotSurvivalProbabilities,* contained in an external file, is called. When it is finished, function *processFigure*, contained in the analysis_process file is called. Finally, function *finish* is invoked to tidy up.

Initializing an analysis

The initialization function performs useful housekeeping. First, it sets a position for the figures based on the information included in the client data structure and the screen size of the computer being used at the time. Then it sets the number for the first figure to 1. Finally, it initializes a stack variable that will store the figure identifiers for the figures previously created that have not been deleted.

For those interested in details, here is the listing.

```
function analysis = initialize( analysis, client )
  % set figure number and initialize stack
      analysis.figNum = 1;
      analysis.stack     = [  ];
end % function initialize
```

Creating a Figure

It is a simple matter to create a new figure in Matlab – just include a *figure* statement requesting one. You can assign the figure an identifier for future reference, but it is sometimes easier to use the global variable *gcf* (get current figure) to reference the one that is currently active.

The *createFigure* function creates a new figure and more. It sets the global colormap to its default (more about this later) and sets the sizes for the fonts of all key figure elements. It also sets the background color to white. Here, as elsewhere, color is indicated by a vector of the proportions of red, blue and green (the three colors utilized on computer screens). This is not the place for a treatise on color theory. Suffice it to say that white is perceived by human beings when all three colors are at their maximum values.

Here is the function, called each time a new figure is to be created.

```
function analysis = createFigure( analysis, client )

 % create a new figure
    fignum = figure;
    set(gcf, 'Position', client.figurePosition );
    analysis.stack = [ analysis.stack  fignum ];

 % set colormap to the default set of colors
    colormap( 'default' );
 % set font sizes
    xl = get( gca, 'Xlabel' );
    set( xl, 'Fontsize', 20 );
    yl = get( gca, 'Ylabel ');
    set( yl, 'Fontsize', 20 );
    ttl = get( gca, 'Title' );
    set( ttl, 'Fontsize', 25 );
    set( gca, 'Fontsize', 20 );
    h = findobj( gcf, 'type', 'text' );
    for i = 1: length(h)
      set( h(i), 'Fontsize', 20 );
    end;
 % set background color
   set(gcf, 'color', [1 1 1] );

% if figures not stacked, remove bottom figure
    if lower( analysis.stackFigures ) == 'n'
      if length( analysis.stack ) > 2
         close( analysis.stack(1) );
         analysis.stack = analysis.stack( 2:length( analysis.stack ) );
      end;

end % function createFigure(  )
```

Processing a Figure

Once a figure has been created, it is processed by calling a function named (not surprisingly) *processFigure*. This changes the number for the next figure, then either pauses the requested number of seconds, if desired; otherwise it beeps and awaits a keypress.

Here is the function:

```
function analysis = processFigure( analysis )

  % change figure number
    analysis.figNum = analysis.figNum + 1;

  % delay before next figure or end
    if analysis.figureDelay > 0
        pause( analysis.figureDelay );
     else
        beep;
        pause;
    end;

end % function analysis = processFigure( analysis )
```

Finishing an Analysis

Only one function in the analysis_process file remains to be described. It is called, appropriately, *finish*. Here it is:

```
function finish( analysis )

    if lower( analysis.stackFigures ) == 'n'
        if length(analysis.stack) > 1
            close( analysis.stack(1) );
        end;
    end;

    if analysis.figuresClosedWhenDone == 'y'
        close all;
    end;

end % function finish(analysis)
```

This is somewhat anticlimactic. If the figures are not to be stacked, the one under the last showing is removed. And if the user wants the figures closed automatically, it shall be done.

Plotting Survival Probabilities

We turn now to the external function that produces a graph of survival probabilities. Here it is in its entirety.

```
function analPlotSurvivalProbabilities( analysis, client, market );
  % plot survival probabilities
  % called by analysis_process function
  % get probabilities of survival
    probSurvive1only =  mean( client.pStatesM == 1 );
    probSurvive2only =  mean( client.pStatesM == 2) ;
    probSurviveBoth =   mean( client.pStatesM == 3 );
    probSurviveAll = [ probSurviveBoth ; probSurvive1only; probSurvive2only ]';
 % create graph
  set( gcf, 'name',  'Recipient Survival Probabilities' );
  set(gcf, 'Position',  analysis.figPosition );
  bar( probSurviveAll, 'stacked' );
  grid on;
  title( 'Recipient Survival Probabilities',  'color',  [0 0 1] );
  xlabel( 'Year' );
  ylabel( 'Probability' );
  legend( 'Both',  [client.p1Name ' only'],  [client.p2Name ' only'] );
  cmap = [ 0 .8 0; 1 0 0;  0 0 1 ];
  colormap( gcf,cmap );
end % plotSurvivalProbabilities(analysis, client,market);
```

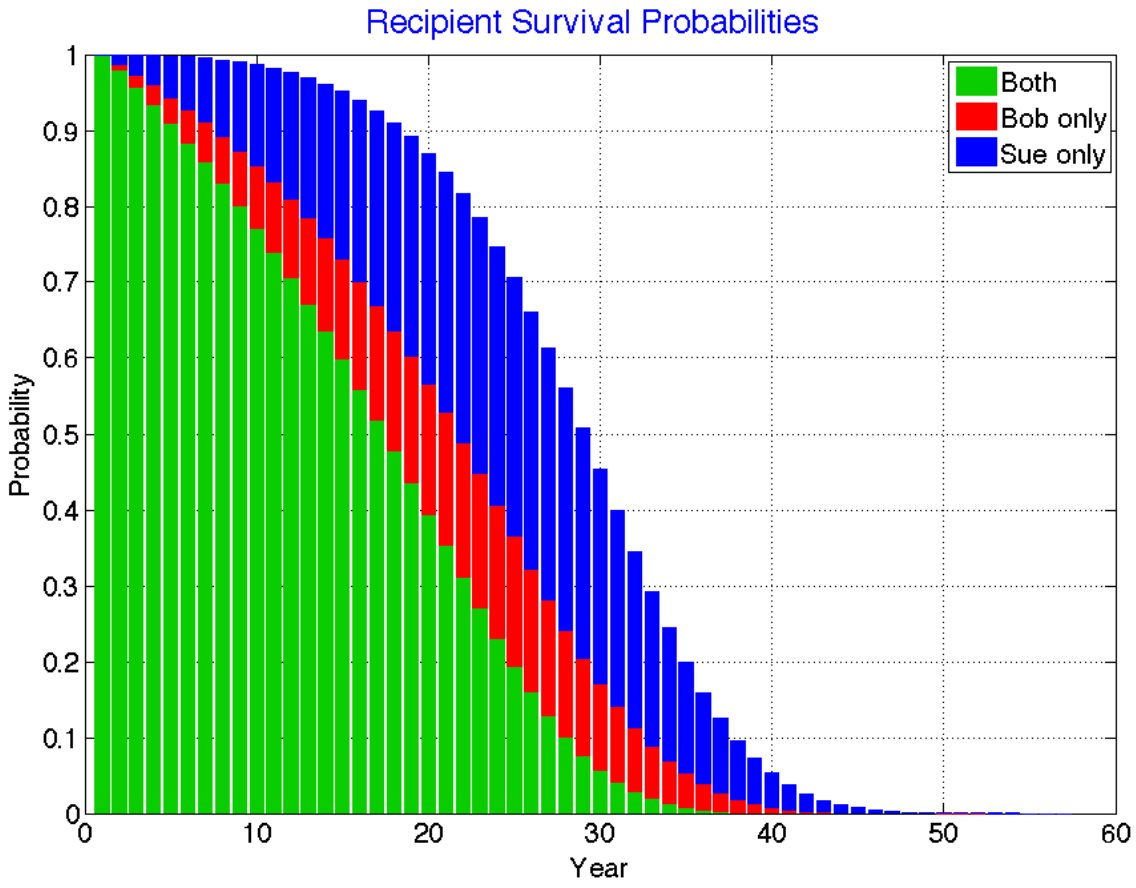
We begin by computing the probabilities of each of the three main personal states, using the means of entries in the *client.pStatesM* matrix for each year. Then these three rows are stacked to form a single matrix, *probSurviveAll*. The rest of the function creates the desired graph. It is given a name and positioned on the screen at the next desired position. Then the Matlab function *bar* is called, with the survival probability matrix as an argument and a request to stack the bars for the three personal states.

The remainder of the statements add elements to the figure. The first turns on a grid. The next adds a title in blue ([0 0 1] for 0 red, 0 green and 1 blue). Labels are provided for the axes, and a legend is added with the client names. Finally, we choose the colors for the bars, with a shade of green ([0 .8 0]) for both recipients, pure red ([1 0 0]) for the first person and pure blue ([0 0 1]) for the second – a color scheme that we will employ in other figures as well. Why not use a "pure" green ([0 1 0]) for "both"? Because it seems washed-out and aesthetically displeasing. Instead we mix green with black ([0 0 0]) to get a deeper color. Idiosyncratic? Perhaps. In any event, this is our choice for the figure's *colormap*.

To produce the survival graph, we need only add the following to the previous case script:

```
% create analysis
 analysis = analysis_create(  );
% select desired output
 analysis.plotSurvivalProbabilities  = 'y';
% process analysis
 analysis_process( analysis, client, market );
```

And here is the graph:



As discussed in Chapter 4, the graph shows the probability of each of the three distinct personal personal states in each year. As we know, Bob's chances for a single life are clearly poorer than Sue's, since he is both older and male. This will affect many of the subsequent analyses as well, as we will see in subsequent chapters.