CHAPTER 4 CREATING A PLOT (USING A LIBRARY MODEL)

In this chapter we are going to produce a plot by using a model that someone else has created. We gain two lessons: (1) you will understand first hand the difficulties one encounters when trying to use a model someone else has created and (2) you will learn how to produce a plot in ASCEND. The approach we take is not the one you should take if your goal is simply to produce this plot. Our goal is pedagogical, not efficiency. In the last chapter we created an array of vessel models to produce the data that we now about to plot. We approached this problem this way so you could see how one creates arrays in ASCEND. Having this model, we have the data. The easiest thing we can do now it use it to produce a plot.

We also have in ASCEND the ability to do case studies over a model instance, varying one or more of the fixed variables for it over a range of values and capturing the values of other variables that result. This powerful case study tool is the proper way to produce this plot as ASCEND only has to compile one instance and solve it repeatedly rather than produce an array of models. We finish this chapter showing you how to use this case study tool.

4.1 CREATING A PLOT

We want a plot of *metal_mass* values vs. *H_to_D_ratio*. If we look around at the available tools, we find there is a *Plot* tool under the *Display* button in the **Browser** window. While not obvious, it turns out we can plot the arrays we produce when we include instances of type *plt_plot_integer* and *plt_plot_symbol* in our model. We find these types in the file *plot.a4l* located in the ASCEND4 models directory which is distributed with ASCEND. Figure 4-1 is a print out of that file (but with line numbers added so we can reference them here).

```
REQUIRE "system.a4l";                                                         1
PROVIDE "plot.a4l";                                                           2
(************************************************************************\    3
   plot.a4l                                                                   4
   by Ben Allan                                                               5
   Part of the Ascend Library
This file is part of the Ascend modeling library.
Copyright (C) 1997 Benjamin Andrew Allan
The Ascend modeling library is free software; you can redistribute
it and/or modify it under the terms of the GNU General Public License as
published by the Free Software Foundation; either version 2 of the
License, or (at your option) any later version.
The Ascend Language Interpreter is distributed in hope that it will be
useful, but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.
You should have received a copy of the GNU General Public License along with
the program; if not, write to the Free Software Foundation, Inc., 675
Mass Ave, Cambridge, MA 02139 USA.  Check the file named COPYING.
\*********************************************************************)
(*********************************************************************\
  $Date: 97/08/04 15:22:21 $
  $Revision: 1.1 $
  $Author: ballan $
  $Source: /afs/cs.cmu.edu/project/ascend/Repository/models/plot.a4l,v $
\*********************************************************************)
(*===========================================================================
==*
    P L O T . A 4 L
    ---------------
   AUTHOR:Ben Allan
      provoked by plot.lib by Peter Piela and Kirk A. Abbott
   DATES:03/97 - Original code.
   CONTENTS:
      A parameterized plot library mostly compatible
      with plot.lib, but with variable graph titles.
*)
MODEL pltmodel() REFINES cmumodel();
END pltmodel;
MODEL plt_point(
   x WILL_BE real;
   y WILL_BE real;
) REFINES pltmodel();
END plt_point;
(***************************************************************)
MODEL plt_curve(
   npnts IS_A set OF integer_constant;
   y_data[npnts] WILL_BE real;
   x_data[npnts] WILL_BE real;
) REFINES pltmodel();
(* points of matching subscript will be plotted in order of
 * increasing subscript value.                                    51
 *)                                                               52
   legend      IS_A symbol; (* mutable now! *)                    53
   FOR i IN [npnts] CREATE                                        54
      pnt[i]IS_A plt_point(x_data[i],y_data[i]);                  55
   END FOR;                                                       56
END plt_curve;                                                    57
(****************************************************************) 58
MODEL plt_plot_integer(                                           59
   curve_set IS_A set OF integer_constant;                        60
   curve[curve_set] WILL_BE plt_curve;                            61
) REFINES pltmodel();                                             62
   title, XLabel, YLabel IS_A symbol; (* mutable now! *)          63
   Xlow IS_A real;                                                64
   Ylow IS_A real;                                                65
   Xhigh IS_A real;                                               66
   Yhigh IS_A real;                                               67
   Xlog IS_A boolean;                                             68
   Ylog IS_A boolean;                                             69
END plt_plot_integer;                                             70
(****************************************************************) 71
MODEL plt_plot_symbol(                                            72
   curve_set IS_A set OF symbol_constant;                         73
   curve[curve_set] WILL_BE plt_curve;                            74
) REFINES pltmodel();                                             75
   title, XLabel, YLabel IS_A symbol; (* mutable now! *)          76
   Xlow IS_A real;                                                77
   Ylow IS_A real;                                                78
   Xhigh IS_A real;                                               79
   Yhigh IS_A real;                                               80
   Xlog IS_A boolean;                                             81
   Ylog IS_A boolean;                                             82
END plt_plot_symbol;                                              83
```

Figure 4-1 The file plot.a4l

As you can see, this file contains the two types we seek—starting in lines 59 and 72, respectively. However, before we can use them, we do need to understand them. We are, so to speak, on the receiving end of the reusability issue. If you spend some time, you will find that you can decipher these model definitions. To make that less painful, we will help you do so here. If these models were better documented, they would be much less difficult to interpret. In time we will add Notes to them to remedy this deficiency.

4.1.1 MODEL REFINEMENT

please, explain “refines”

The first model, pltmodel, is two lines long, having a *MODEL* statement indicating it “refines” *cmumodel* and an *END* statement. We have not encountered the concept of refinement as yet. In ASCEND the “refines” means the model *pltmodel* inherits all the statements of *cmumodel*, a model which has been defined at the end of the file *system.a4l*. We show the code for *cmumodel* in Figure 4-2, and we note that it too is an empty model. It is, as it says, a root for a collection of loosely related models. You will note (and forgive) a bit of dry humor by its author, Ben Allan. So far as we know, this model neither provokes nor hides any bugs.

```
MODEL cmumodel();
(* This MODEL does nothing except provide a root
 * for a collection of loosely related models.
 * If it happens to reveal a few bugs in the software,
 * and perhaps masks others, well, what me worry?
 * BAA, 8/97.
 *)
END cmumodel;
```

Figure 4-2 The code for cmumodel

We need to introduce the concept of type refinement to understand these models. We divert for a moment to do just that.

parents and children in a refinement hierarchy

Suppose model *B* refines model *A*. We call *A* the parent model and *B* the child. The child model *B* inherits all the code defining the parent model *A*. In writing the code for model *B*, we do not write the code it inherits from *A*; we simply understand it is there already. The code we write for model *B* will be only those statements that we wish to add beyond the code defining its parent. ASCEND supports only single inheritance; thus a child may have only one parent. A parent, on the other hand, may have many children, each inheriting its code.

order does not matter in declarative code

We are dealing in ASCEND with models defined by their variables and equations. As we have noted above, the order for the statements defining each of these does not matter—i.e., the variables and equations may be defined in any order. So adding new variables and equations through refinement may be done quite easily.

but it does in the procedural code for methods

In contrast, the methods are bits of procedural code—i.e., they are run as a sequence of statements where order does matter. In ASCEND, a child model will inherit all the methods of the parent. If you wish to alter the code for a method, you must replace it entirely, giving it the same name as the method to be replaced. (However, if you look into the documentation on the methods (syntax.pdf), you will find that the original method is still available for execution. You simply have to add a qualifier to its name to point to it.)

If we look into this file we see the refinement hierarchy shown in Figure 4-3. *cmumodel* is the parent model for all these models. *pltmodel* is its child. The remaining three models are children of *pltmodel*.

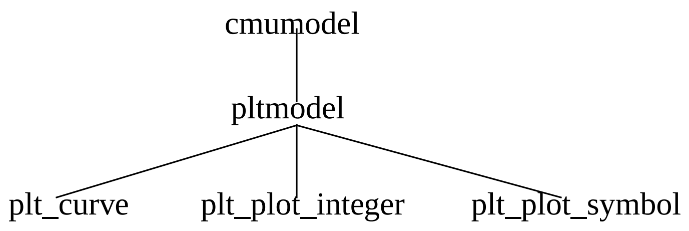

Figure 4-3 The refinement hierarchy in the file plot.a4l

(We can have ASCEND show us the refinement hierarchy. From the **Library** window, select *Read types from file* from the *File* button, and click on *plot.a4l* (you may need to change the filter to see the *.a4l* files). Select *plot.a4l* from the left hand-pane of the **Library**, and then *plt_plot_symbol* from the right-hand pane. Finally, choose the *Ancestry* tool from the *Display* button.)

There are three reasons to support model refinement, with the last being the most important one.

reasons for refinement

- **We write more compact code**: The first reason is compactness of coding. One can inherit a lot of code from a parent. Only the new statements belonging to the child are then written to define it. This is not a very important reason for having refinement.
- **Changes we make to the parent propagate**: A second reason is that one can edit changes into the parent and know that the children will inherit those changes without having to alter the code written for the child. (Of course, one can change the parent in such a way that the changes to the child are not what is wanted for the child, introducing what will likely become some interesting debugging problems.)

with the most important being we know what can substitute for what

- **We know what can substitute for what**: The most important reason is that inheritance tells us what kinds of parts may be substituted for a particular part in a model. Because a child inherits all the code from its parent, we know the child has all the variables and equations defined for it that the parent does—and typically more. **We can use an instance of the child as a replacement for an instance of the parent.** Thus if you were to write a model with the part *A1* of type *A* in it, someone else can create an instance of your model and substitute a part *B1* which is of type *B*. This substituted part will have all the needed variables in it that you assumed would be there.

This third reason says that when a object passed as a parameter WILL_BE of type *A*, we know that a part of either type *A* or type *B* will work.

4.1.2 CONTINUING WITH CREATING A PLOT

We are going to include in our model a part of type *plt_plot_integer* or *plt_plot_symbol* that ASCEND can plot. We need to look at the types of parameters required by whichever of these two we select to include here. Tracing back to its parents, we see them to be empty so all the code for these types is right here.

The first parameter we need is a *curve_set* which is defined to be a set of *integer_constant* or of *symbol_constant*. We have to guess at this time at the purpose for *curve_set*. It would really help to have notes defining the intention here and to have a piece of code that would demonstrate the use of these models. At present, we do not. We proceed, admitting we will appear to “know” more than we should about this model. It turns out that *curve_set* allows us to identify each of the curves we are going to plot. These models assume we are plotting several variables (let’s call them *y[1]*, *y[2]*, ...) against the same independent variable *x*. The values for curve_set are the ‘1’, ‘2’, etc. identifying these curves.

Here we wish to plot only one curve presenting *metal_mass* vs. *H_to_D_ratio*. We can elect to use *plt_plot_symbol* and label this curve ‘*5 mm*’. The label ‘*5 mm*’ is a *symbol* so we will create a set of type *symbol* with this single member.

The second object has to be a object of type *plt_curve*.

Looking at line 45, we see how to include an object of type *plt_curve*. It must be passed three objects: a set of integers (e.g., the set of integers from *1* to *20*) and two lists of data giving the *y*-values vs. the *x*-values for the curve. In the model *tabulated_vessel_values*, we have just these two lists, and they are named *metal_mass* and *H_to_D_ratio*.

In Figure 4-4, we show the code you need to add to the model *tabulated_vessel_values*. It contains a part called *massVSratio* of type *plt_plot_symbol* that ASCEND can plot. This code is at the end of the declarative statements in tabulated_vessel_values. It also replaces the first method, METHOD default_self.

```
    CurveSet "the index set for all the curves to be plotted"
            IS_A set OF symbol_constant;
    CurveSet :== ['5 mm'];

    Curves['5 mm'] "the one curve of 20 points for metal_mass vs. H_to_D_ratio"
            IS_A plt_curve([1..n_entries], metal_mass, H_to_D_ratio);
    massVSratio "the object ASCEND can plot"
            IS_A plt_plot_symbol(CurveSet, Curves);

METHODS
METHOD default_self;
(* set the title for the plot and the labels for the ordinate and abscissa *)
    massVSratio.title :=
     'Metal mass of the walls vs H to D ratio for a thin-walled cylindrical 
vessel';
    massVSratio.XLabel := 'H to D ratio';
    massVSratio.YLabel := 'metal mass IN kg/m^3';
END default_self;
```

Figure 4-4 The last bit of new code to include a plot in the model *tabulated_vessel_values* (save as vesselPlot.a4c)

Also just after the first line in this file—which reads

```
REQUIRE “atoms.a4l”;
```

place the instruction

```
REQUIRE “plot.a4l”;
```

When you solve this new instance and make *massSVratio* the current object, you will find the *Plot* tool under the *Display* button in the **Browser** window lights up and can be selected. If you do this, you will get a plot of *metal_mass* vs. *H_to_D_ratio*. A clear minimum is apparent on this plot at *H_to_D_ratio* equal to approximately one.

You should create a script to run this model just as you did for *vesselTabulated.a4c* in the previous chapter. Save it as *vesselPlot.a4s*.

4.2 CREATING A CASE STUDY FROM A SINGLE VESSEL

You may think creating an array of vessels and a complex plot object just to generate a graph is either an awful lot of work or a method which will not work for very large models. You think correctly on both points. The plt_plot models are primarily useful for sampling values from an array of inter-related models that represent a spatially distributed system such as the pillars in a bridge or the trays in a distillation column. You can conduct a case study, solving a single model over a range of values for some specified variable, using the Script command STUDY.

We will step through creating a base case and a case study using the vessel model. Start by opening a new buffer in the Script window and turning on the record button of the Script's edit menu. In the Library window run the "Delete all types" button to clear out any previous simulations. Load the vessel model from the file vesselMethods.a4c you created in Section 3.2.

4.2.1 THE BASE CASE

compile a vessel.

Select and compile the vessel model. Give the simulation the name `V`. Select the simulation `V` in the bottom pane of the Library window and use the right mouse button (or Alt-x b) to send the simulation to the Browser.

solving the base case.

In the Browser, place the mouse cursor over the upper left pane. Use the right mouse button to run methods `reset` and `values`, then send the model to the Solver by typing "Alt-x s". Move the mouse to the Solver window and hit the F5 key to solve the model.

graphical case study optimization

We now know that it takes 535.7 kg of metal to make a 250 cubic foot vessel which is twice as high as it is broad. Suppose that now we want to know the largest volume that this amount of metal can contain assuming the same wall thickness is required. Perhaps a skinnier or fatter vessel can hold more, so we need to do a case study using the aspect ratio (`H_to_D_ratio`) as the independent variable. Use the Browser to change `V.metal_mass.fixed` to TRUE, since we are using a constant amount of metal. The solver will want you to free a variable now, so select `V.vessel_vol` to be freed, since volume is what we want to study.

script recorded so far

Turn off the recording button on the Script window. The recording should look something like

```
DELETE TYPES;
READ FILE {vesselMethods.a4c};
COMPILE V OF vessel;
BROWSE {V};
RUN {V.reset};
SOLVE {V} WITH QRSlv;
ASSIGN {V.metal_mass.fixed} TRUE {};

# you must type the next line in the script yourself.
ASSIGN {V.vessel_vol.fixed} FALSE {};
```

The file ascend4/models/vesselStudy.a4s was recorded in a similar manner.

4.2.2 CASE STUDY EXAMPLES

configuring a case study

The STUDY command takes a lot of arguments. We'll explain them all momentarily, but should you forget them simply enter the command `STUDY` without arguments in the ASCEND Console window or xterm window to see an error message explaining the arguments and giving an example. Enter the following command in the Script window exactly as shown except for the file name following `OUTFILE`. Specify a file to be created in *your* ascdata directory.

```
STUDY {vessel_vol} \
IN {V} \
VARYING {{H_to_D_ratio} {0.1} {0.5} {0.8} {1} {1.5} {2} \
{3} {4} {8}} \
USING {QRSlv} \
OUTFILE {/usr0/ballan/ascdata/vvstudy.dat} \
ERROR STOP;
```

This is the simplest form of case study; the backslashes at the end of each line mean that it is all one big statement. Select all these lines in the Script at once with the mouse and then hit F5 to execute the study. The solver will solve all the cases and produce the output file vvstudy.dat. The quickest way to see the result is to enter the following command in the Script, then select and execute it. (Remember to use the name of your file and not the name shown).

```
ASCPLOT {/usr0/ballan/ascdata/vvstudy.dat};
ASCPLOT CLOSE; #omit if you want to see data table
```

You should get a graph that looks something like Figure 4-5. The largest volume is in the neighborhood of an `H_to_D_ratio` of 1.

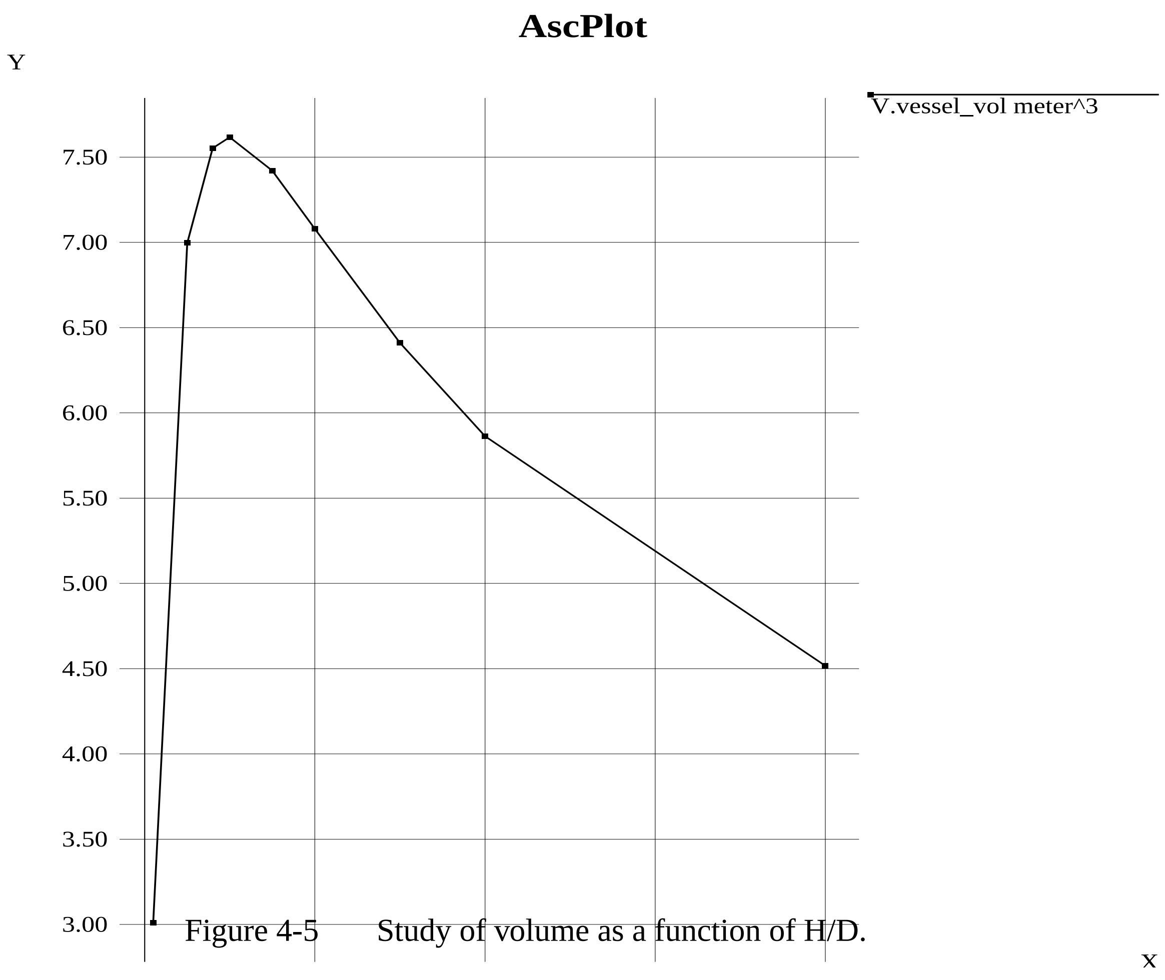

Figure 4-5 Study of volume as a function of H/D.

4.2.2.1 MULTI-VARIABLE STUDIES

We now have an idea where the solution is most interesting, so we can do a detailed study where we also monitor other variables such as surface areas. Additional variables to watch can be added to the STUDY clause of the statement.

```
STUDY {vessel_vol} {end_area} {side_area} \
IN {V} \
VARYING {{H_to_D_ratio} {0.5} {0.6} {0.7} {0.8) {0.9} \
{1} {1.1} {1.2} {1.3}} \
USING {QRSlv} \
OUTFILE {/usr0/ballan/ascdata/vvstudy.dat} \
ERROR STOP;
ASCPLOT {/usr0/ballan/ascdata/vvstudy.dat};
ASCPLOT CLOSE; #omit if you want to see data table
```

4.2.2.2 MULTI-PARAMETER STUDIES

We can also do a multi-parameter study, for example also varying the wall thickness allowed. In general, any number of the fixed variables can be varied in a single study, but be aware that ASCEND's relatively simple plotting capabilities do not yet include surface or contour maps so you will need another graphic tool to view really pretty pictures.

```
STUDY {vessel_vol} \
IN {V} \
VARYING \
{{H_to_D_ratio} {0.8) {0.9} {1} {1.1} {1.2} {1.3}} \
{{wall_thickness} {4 {mm}} {5 {mm}} {6 {mm}} {7 {mm}}} \
USING {QRSlv} \
OUTFILE {/usr0/ballan/ascdata/vvstudy.dat} \
ERROR STOP;
ASCPLOT {/usr0/ballan/ascdata/vvstudy.dat};
```

In this study the peak volume occurs at the same `H_to_D_ratio` for any wall thickness but the vessel volume increases for thinner walls. This may be hard to see with the default graph settings, but column 2 in rows 8-11 (H_to_D = 1.0) of the ASCPLOT data table have the largest volumes for any given thickness in column 1. Notice that the units must be specified for the `wall_thickness` values in the VARYING clause.

4.2.2.3 PLOTTING OUTPUT WITH OTHER TOOLS

To convert the study results from the ASCPLOT format to a file more suitable for importing into a spreadsheet, the following command does the trick. As usual, change the names to match your `ascdata` directory.

```
asc_merge_data_files excel \
{/usr0/ballan/ascdata/vvs.txt} \
{/usr0/ballan/ascdata/vvstudy.dat}
```

If you prefer Matlab style text, substitute 'matlab' for 'excel' in the line above and change the output name from 'vvs.txt' to 'vvs.m'.

4.2.3 STUDY BEHAVIOR DETAILS

variable list

We now turn to the details of the STUDY statement. As we saw in Section 4.2.2.1, any number of variables to be monitored can follow the STUDY keyword.

IN clause

The IN clause specifies which part of a simulation is to be sent to the Solver; a small part of a much larger model can be studied if you so desire. All the variable and parameter names that follow the STUDY keyword and that appear in the VARYING clause must be found in this part of the simulation.

parameter list

The VARYING clauses is a list of lists. Each inner list gives the name of the parameter to vary followed by its list of values. Each possible combination of parameter values will be attempted in multi-parameter studies. If a case fails to solve, then the study will behave according to the option set in the ERROR clause.

solver name

The solver named in the USING clause is invoked on each case. The solver may be any of the algebraic solvers or optimizers, but the integrators (e.g. LSODE) are not allowed.

data file name

The case data are stored in the file name which appears in the OUTFILE clause. By default, this file is overwritten when a STUDY is started, so if you want multiple result files, use separate file names.

error handling

When the solver fails to converge or encounters an error, the STUDY can either ignore it (`ERROR IGNORE`) and go on to the next case, warn you (`ERROR WARN`) and go on to the next case, or stop (`ERROR STOP`). The ERROR option makes it possible start a case study and go to lunch. Cases which fail to solve will not appear in the output data file.

Note that if the model is numerically ill-behaved it is possible for a case to fail when there is in fact a solution for that combination of parameters. STUDY uses the solution of the last successfully solved case as the initial guess for the next case, but sometimes this is not the best strategy. STUDY also does not attempt to rescale the problem from case to case. When a case that you think should succeed fails, go back and investigate that region of the model again manually or with a more narrowly defined study.

4.3 DISCUSSION

We have just led you step by step through the process of creating, debugging and solving a small ASCEND model. We then showed you how to make this model more reusable, first by adding comments and methods. Methods capture the “how you got it well-posed” experience you had when first solving an instance of the vessel model. We then showed you how to parameterize this model and then use it to construct a table of *metal_mass* values vs. *H_to_D_ratio* values. Finally we showed you how to add a plot of these results. You should next look at the chapter in the documentation where you create two more small ASCEND models. This chapter gives you much less detail on the buttons to push. Finally, if you are a chemical engineer, you should look at the chapter on the script and model for a simple flowsheet (simple_fs.a4s and simple_fs.a4c respectively).

With this experience you should be ready to write your own simple ASCEND models to solve problems that you might now think of solving using a spreadsheet. Remember that once you have the model debugged in ASCEND, you can solve inside out, backwards and upside down and NOT just the way you first posed it—unlike your typical use of a spreadsheet model.