October 9, 1994

The ASCEND modeling language

Molar stream example

We shall start our description of the ASCEND modeling language by presenting some examples. Let us start with the model of what superficially seems a simple concept: a stream characterized only by the flows for the components in it. We will build on this model to construct that for some simple unit operations and then for a total flowsheet.

Molar stream

To model a simple molar stream, we must decide on the concepts on which it is to be constructed. A molar stream is one for which one lists only the molars flowrate and the mole fraction for each of the components in it as well as a total flowrate.

We shall apply some hard learned knowledge about modeling streams and insist that the intensive properties for a stream be collected together into a single concept we shall call its state. Then a stream will be its state plus its extensive properties. Intensive variables are those that are not quantity dependent. Here the only intensive variables are those describing the composition of the stream, i.e., the mole fractions for each of the components. The molar flowrates and the total molar flowrate are extensive variables whose values would double if we were to double the size of the stream.

We will call this collection of intensive variables for a molar stream a *molar mixture*. This model is in Fig. 1. In it we define a set of mole fractions, one for each member of a set of component names.

Model structure

Declarative

This model has a number of statements in it that we need to discuss. It is headed by a line stating we are defining a new model for ASCEND. The next two lines define the parts in the model and what type of thing each is. One item is a set of components names, each of which is of type *symbol*. A symbol is a built-in type in ASCEND; it is an unbroken sequence of up to 36 alphanumeric characters enclosed within two single quotes. For each member `i` in this set of components, we define a variable `y[i]` whose interpretation is to be the mole fraction for that species in the stream. It is of type *fraction*. The definition for fraction is in the library file called atoms.lib which is available as a separate document. Atoms.lib contains all definitions for a large number of different variable types and should be referenced before starting the writing of a model.

Fig. 2 shows a typical variable type definition, one for molar_rate. One can define a new type of variable at any time by including a new *atom* definition. An atom definition may be thought of as the most elementary model definition in ASCEND and is used solely to define variable types. All other type definitions in ASCEND are called *models*.

The definition for the type *molar_rate* illustrates the idea of inheritance, a key capability of the ASCEND language. We see that molar_rate is a refinement of a previously defined type *solver_var*. Basically one can define a model or an atom to inherit all the statements of a previously defined model or atom and then add further statements to complete the new type definition. Thus the new item is everything the older one is plus more.

```
MODEL mixture;

   components                            IS_A  set OF symbol;
        y[components]                    IS_A  fraction;

        SUM(y[i] | i IN components) = 1.0;

  INITIALIZATION
    PROCEDURE clear;
        y[components].fixed              := FALSE;
    END clear;

    PROCEDURE specify;
        y[components].fixed              := TRUE;
        y[CHOICE(components)].fixed      := FALSE;
    END specify;

    PROCEDURE reset;
        RUN clear;
        RUN specify;
    END reset;

END mixture;
```

Figure 1. An ASCEND model for the intensive variables in a molar stream

```
ATOM molar_rate REFINES solver_var
      DIMENSION Q/T
      DEFAULT 100.0{lb_mole/hour};
      lower_bound                        := 0.0{lb_mole/hour};
      upper_bound                        := 1e50{lb_mole/hour};
      nominal                            := 100.0{lb_mole/hour};
END molar_rate;
```

Figure 2. Definition of the variable type *molar_rate*

Returning to our mixture model, the mole fractions must add to unity. We add this constraint using the SUM function which is built into ASCEND. The sum function adds together all of its arguments; there is no limit on the number of arguments it may have.

Procedural

So far everything is simple. We now see a section of the model entitled INITIALIZATION, and, in this section, we find several PROCEDURE definitions. Any set of procedures may be included. A modeler can ask that the ASCEND system to execute these procedures at any time and in any order while debugging and solving. They are executed ONLY on request. They are true procedures; the statements execute in order and the value of a variable on the left-hand-side of a statement is replaced by the value of the expression on the right.

The particular procedures included here are based on experience and are becoming a standard for all models we write. Let us make this standard into a modeling rule as follows.

Good modeling practice in ASCEND: Attach procedures *clear*, *specify* and *reset* (and *scale*) to all models created in ASCEND as a part of good modeling practice.

Doing so will force a modeler to consider these issues carefully. Also, from our experience in creating and debugging models, these issues, once resolved, are an excellent starting place for all other uses of a model.

Associated with every computable variable in ASCEND is a boolean flag indicating if its value is to be fixed or not when solving. If *TRUE*, the variable value is held at its current value while solving; if *FALSE* its value is computed using the equations defining the model. In the *clear* procedure we set the flags for all computable variables to *FALSE*.

In the *specify* procedure, we set exactly enough of the flags to TRUE to make the model square (i.e., *n* equations in *n* unknowns). This model has exactly one equation. The fixed flag for each of the compositions is set to TRUE except for one which can be computed. The CHOICE function selects exactly one of the elements in the set mentioned, the choice being arbitrary.

The *reset* procedure runs the clear procedure and then the specify procedure to return the model fixed flags to a known standard state.

Finally, we use the *scale* procedure to allow extensive variables to be scaled according to their current values. This model only contains intensive variables so the *scale* procedure is empty. As a part of good modeling practice, we do insist that the procedure exists even if it is empty. Scaling resets the values for the nominal attribute of a variable. Do not set this value to zero or leave it at zero when, for example, defining a new type of variable.

Molar stream example

We are now ready to create the molar stream model, Fig. 3.

The molar stream model contains a part called state whose type is a molar mixture, a type we just defined. It also has variables to represent the total molar flowrate for the stream and the molar flows for each of the stream components.

We see our first ARE_THE_SAME statement. The set of component names defined for the stream and the set of component names inside the state are to be exactly the same set. In the current version of ASCEND, these sets are stored in the exact same storage locations. The one set has two names in the molar stream model: *components* and *state.components*.

```
MODEL molar_stream;

      components                          IS_A  set OF symbol;
      state                               IS_A  mixture;
      Ftot,f[components]                  IS_A  molar_rate;

      components, state.components        ARE_THE_SAME;


      FOR i IN components CREATE
         f_def[i]: f[i] = Ftot*state.y[i];
      END;

   INITIALIZATION

      PROCEDURE clear;
         RUN state.clear;
         Ftot.fixed                       := FALSE;
         f[components].fixed              := FALSE;
      END clear;

      PROCEDURE seqmod;
         RUN state.specify;
         state.y[components].fixed        := FALSE;
      END seqmod;

      PROCEDURE specify;
         RUN seqmod;
         f[components].fixed              := TRUE;
      END specify;

      PROCEDURE reset;
         RUN clear;
         RUN specify;
      END reset;

      PROCEDURE scale;
         FOR i IN components DO
            f[i].nominal                  := f[i] + 0.1{mol/s};
         END;
         Ftot.nominal                     := Ftot + 0.1{mol/s};
      END scale;

END molar_stream;
```

Figure 3. An ASCEND model for a molar stream

The approach is that one can attach to any instance within a part by constructing a qualified name for it. The name has the structure:

part_name.inner_instance_name

Qualified names

ASCEND allows access into parts within a model through the use of qualified names. We placed a design requirement on the language that the users of a model definition would define the interface to that model; i.e., users, and not the model creator, would be able to select which variables within a model they wish to access. In this manner the same model definition can satisfy different uses without itself being touched.

Such a naming scheme can be repeated to any depth, allowing us to go, so to speak, to the bottom-most layers of a model and use an instance appearing there at the outer-most level.

We used qualified names in the model *molar_stream* above. This model contains a part called *state* which is an instance of the model *mixture*. Mixture has a part called *components*, the set of names for the components in it. Molar_stream also has a set called *components*, and we wanted these two sets to be the same set. We needed a means to name the set called components within the part called state which we did by constructing the name

```
state.components
```

and using it in an ARE_THE_SAME statement.

In the procedures, we accessed the fixed attribute for variables when we wanted to select which variables are to be fixed during a computation. An example was

```
f[components].fixed                :=    TRUE;
```

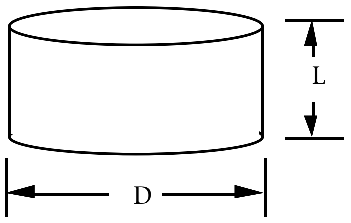
which is interpreted as fixing the molar flowrates for the whole set of components.

Constant relative volatility flash example

Fig. 4 is a simple relative volatility flash model. It uses the previously defined concept of a stream in its definition. It could be interesting to compare the equations it will generate to the ten we listed for the simple two component model earlier.

Homework 8

1. Create an ASCEND model to compute the mass of metal in a thin walled cylindrical tank shown the following figure. The tank has flat ends. Wall thickness is t.



2. Write an ASCEND model for the two-species flash model we used earlier to describe modeling. Include all parts for it.

```
MODEL flash;

      feed,vap,liq                        IS_A  molar_stream;

      feed.components,
         vap.components,
         liq.components                   ARE_THE_SAME;

      alpha[feed.components],
         ave_alpha                        IS_A  factor;

      vap_to_feed_ratio                   IS_A  fraction;

      vap_to_feed_ratio*feed.Ftot = vap.Ftot;

      FOR i IN feed.components CREATE
         cmb[i]: feed.f[i] = vap.f[i] + liq.f[i];
         eq[i]:  vap.state.y[i]*ave_alpha = alpha[i]*liq.state.y[i];
      END;

   INITIALIZATION

      PROCEDURE clear;
         RUN feed.clear;
         RUN vap.clear;
         RUN liq.clear;
         alpha[feed.components].fixed     :=    FALSE;
         ave_alpha.fixed                  :=    FALSE;
         vap_to_feed_ratio.fixed          :=    FALSE;
      END clear;

      PROCEDURE seqmod;
         alpha[feed.components].fixed     :=    TRUE;
         vap_to_feed_ratio.fixed          :=    TRUE;
      END seqmod;

      PROCEDURE specify;
         RUN seqmod;
         RUN feed.specify;
      END specify;

      PROCEDURE reset;
         RUN clear;
         RUN specify;
      END reset;

      PROCEDURE scale;
         RUN feed.scale;
         RUN vap.scale;
         RUN liq.scale;
      END scale;

END flash;
```

Figure 4. Simple flash model written in the ASCEND language

Complete simple flowsheet example

Fig. 5a-h gives the code (plus the models *mixture*, *molar_stream* and *flash* above) for an entire simple flowsheet. We add models for a *mixer*, a *reactor* and a *splitter*. The model *flowsheet* configures the entire flowsheet, shown in Fig. 5a. Test routines complete the model. We also add a computational controller to allow us the specify the degree of conversion for the reaction in the reactor in terms of component B.

Note that comments in ASCEND start with *(** and end with **)*. You should be able to read the code for this model and understand it without further explanation.

Merging to configure complex models

Another important concept in modeling is the ability to configure a complex model by putting a number of simpler parts together to form the model. The model called *flowsheet* in Fig. 5e illustrates the power of this form of configuration. It declares first that there are to be four parts to the flowsheet: a mixer called m1, a reactor called r1, a flash called fl1 and a splitter called sp1. The mixer is to have two inputs while the splitter has two outputs. We configure the flowsheet by *merging* streams using the ASCEND operator called ARE_THE_SAME. The output stream from the mixer is exactly the same stream as the feed to the reactor. Making them the same is done in ASCEND during compilation by assigning the same storage locations for the same variables in both streams. Also, ASCEND allows only one of the two models to create its equations to avoid the production of redundant equations.

```
(*
The following example illustrates equation based modeling using the
ASCEND system.  The process is a simple recycle process.



                                          -------
                                         |       |
                   ----------------------| split |----> purge
                  |                      |       |
                  |                       -------
                  |                          ^
                  v                          |
                -----      ---------      -------
               |     |    |         |    |       |
         ----->| mix |--->| reactor |--->| flash |
               |     |    |         |    |       |
                -----      ---------      -------
                                             |
                                             |
                                              ----->  C

This model requires:    "system.lib"
                        "atoms.lib"
*)
```

Figure 5a. Diagram of simple flowsheet for simple flowsheet example

```
MODEL mixer;

      n_inputs                          IS_A  integer;
      feed[1..n_inputs], out            IS_A  molar_stream;

      feed[1..n_inputs].components,
         out.components                 ARE_THE_SAME;

      FOR i IN out.components CREATE
         cmb[i]: out.f[i] = SUM(feed[1..n_inputs].f[i]);
      END;

   INITIALIZATION

      PROCEDURE clear;
         RUN feed[1..n_inputs].clear;
         RUN out.clear;
      END clear;

      PROCEDURE seqmod;
      END seqmod;

      PROCEDURE specify;
         RUN seqmod;
         RUN feed[1..n_inputs].specify;
      END specify;

      PROCEDURE reset;
         RUN clear;
         RUN specify;
      END reset;

      PROCEDURE scale;
         RUN feed[1..n_inputs].scale;
         RUN out.scale;
      END scale;

END mixer;
```

Figure 5b. Mixer model for flowsheet example

```
MODEL reactor;

      feed, out                          IS_A  molar_stream;
      feed.components, out.components    ARE_THE_SAME;

      turnover                           IS_A  molar_rate;
      stoich_coef[feed.components]       IS_A  factor;

      FOR i IN feed.components CREATE
         out.f[i] = feed.f[i] + stoich_coef[i]*turnover;
      END;

   INITIALIZATION

      PROCEDURE clear;
         RUN feed.clear;
         RUN out.clear;
         turnover.fixed                          :=    FALSE;
         stoich_coef[feed.components].fixed      :=    FALSE;
      END clear;

      PROCEDURE seqmod;
         turnover.fixed                          :=    TRUE;
         stoich_coef[feed.components].fixed      :=    TRUE;
      END seqmod;

      PROCEDURE specify;
         RUN seqmod;
         RUN feed.specify;
      END specify;

      PROCEDURE reset;
         RUN clear;
         RUN specify;
      END reset;

      PROCEDURE scale;
         RUN feed.scale;
         RUN out.scale;
         turnover.nominal := turnover.nominal+0.0001 {kg_mole/s};
      END scale;

END reactor;
```

Figure 5c. Reactor model for simple flowsheet example

```
 MODEL splitter;

      n_outputs                           IS_A  integer;
      feed, out[1..n_outputs]             IS_A  molar_stream;
      split[1..n_outputs]                 IS_A  fraction;

      feed.components,
         out[1..n_outputs].components     ARE_THE_SAME;

      feed.state,
         out[1..n_outputs].state          ARE_THE_SAME;

      FOR j IN [1..n_outputs] CREATE
         out[j].Ftot = split[j]*feed.Ftot;
      END;

      SUM(split[1..n_outputs]) = 1.0;

   INITIALIZATION

      PROCEDURE clear;
         RUN feed.clear;
         RUN out[1..n_outputs].clear;
         split[1..n_outputs-1].fixed      :=    FALSE;
      END clear;

      PROCEDURE seqmod;
         split[1..n_outputs-1].fixed      :=    TRUE;
      END seqmod;

      PROCEDURE specify;
         RUN seqmod;
         RUN feed.specify;
      END specify;

      PROCEDURE reset;
         RUN clear;
         RUN specify;
      END reset;

      PROCEDURE scale;
         RUN feed.scale;
         RUN out[1..n_outputs].scale;
      END scale;

END splitter;
```

Figure 5d. Splitter model for simple flowsheet example

```
MODEL flowsheet;

      m1                                    IS_A  mixer;
      r1                                    IS_A  reactor;
      fl1                                   IS_A  flash;
      sp1                                   IS_A  splitter;

(* define sets *)

      m1.n_inputs                           :=    2;
      sp1.n_outputs                         :=    2;

(* wire up flowsheet *)

      m1.out, r1.feed                       ARE_THE_SAME;
      r1.out, fl1.feed                      ARE_THE_SAME;
      fl1.vap, sp1.feed                     ARE_THE_SAME;
      sp1.out[2], m1.feed[2]                ARE_THE_SAME;

   INITIALIZATION

      PROCEDURE clear;
         RUN m1.clear;
         RUN r1.clear;
         RUN fl1.clear;
         RUN sp1.clear;
      END clear;

      PROCEDURE seqmod;
         RUN m1.seqmod;
         RUN r1.seqmod;
         RUN fl1.seqmod;
         RUN sp1.seqmod;
      END seqmod;

      PROCEDURE specify;
         RUN seqmod;
         RUN m1.feed[1].specify;
      END specify;

      PROCEDURE reset;
         RUN clear;
         RUN specify;
      END reset;

      PROCEDURE scale;
         RUN m1.scale;
         RUN r1.scale;
         RUN fl1.scale;
         RUN sp1.scale;
      END scale;

END flowsheet;
```

Figure 5e. Code for simple flowsheet

```
MODEL controller;

      fs                                    IS_A  flowsheet;
      conv                                  IS_A  fraction;
      key_components                        IS_A  symbol;
      fs.r1.out.f[key_components] = (1 -
            conv)*fs.r1.feed.f[key_components];

   INITIALIZATION

      PROCEDURE clear;
         RUN fs.clear;
         conv.fixed                         :=    FALSE;
      END clear;

      PROCEDURE specify;
         RUN fs.specify;
         fs.r1.turnover.fixed               :=    FALSE;
         conv.fixed                         :=    TRUE;
      END specify;

      PROCEDURE reset;
         RUN clear;
         RUN specify;
      END reset;

      PROCEDURE scale;
         RUN fs.scale;
      END scale;

END controller;
```

Figure 5f. Code for controller

```
MODEL test_flowsheet REFINES flowsheet;

      m1.out.components                   :=    ['A','B','C'];

   INITIALIZATION

      PROCEDURE values;
         m1.feed[1].f['A']                :=    0.005 {kg_mole/s};
         m1.feed[1].f['B']                :=    0.095 {kg_mole/s};
         m1.feed[1].f['C']                :=    0.0 {mole/s};

         r1.stoich_coef['A']              :=    0;
         r1.stoich_coef['B']              :=    -1;
         r1.stoich_coef['C']              :=    1;

         fl1.alpha['A']                   :=    12.0;
         fl1.alpha['B']                   :=    10.0;
         fl1.alpha['C']                   :=    1.0;
         fl1.vap_to_feed_ratio            :=    0.9;
         fl1.ave_alpha                    :=    5.0;

         sp1.split[1]                     :=    0.01;

         fl1.liq.Ftot                     :=    m1.feed[1].f['B'];
      END values;

END test_flowsheet;
```

Figure 5g. Test code for simple flowsheet

```
MODEL test_controller REFINES controller;

      fs       IS_REFINED_TO              test_flowsheet;
      key_components                      :=    'B';

   INITIALIZATION

      PROCEDURE values;
         RUN fs.values;
         conv                             :=    0.07;
      END values;

END test_controller;
```

Figure 5h. Code to test controller

Inheritance and deferred binding in modeling

The flowsheet example illustrates inheritance (using the REFINES operator) in ASCEND. It does not illustrate deferred binding, another of the modeling concepts we feel is necessary in equation-based modeling. Model and atom definitions in ASCEND can establish a refinement hierarchy of definitions. For example, Fig. 6 gives the code for five related models. There is a model hierarchy implied in these statements which we can express either using the indented format of Fig. 6 or in graphical form as shown in Fig. 7. The indenting in Fig. 6 is strictly a matter of style in displaying the code.

```
MODEL a;
   statements defining model a
END a;

   MODEL b REFINES a;
     statements defining model b
   END b;

      MODEL c REFINES a;
        statements defining model c
      END d;

   MODEL d REFINES a;
     statements defining model d
   END d;

MODEL e;
  statements defining model e
END e;
```

Figure 6. Sample code to establish a model hierarchy

In the hierarchy in Figs. 6 and 7, *c* has both *b* and *a* as *ancestors* in the refinement hierarchy. *d* also has *a* as an ancestor, but it is a refinement which is *incompatible* with *b* and *c*. We use inheritance in ASCEND for two reasons. **First**, it is efficient to be able to state that a model contains all the code used to define another. One avoids rewriting the repeated code. This use is not the important reason, however.
Second, and much more important, is that we know that models *b* and *c* are everything model *a* is and more; thus any model that has a part of type *a* in it can use a part of type *b* or type *c* as the actual part when we compile the model.

Changing the type of a part is called *deferred binding* (using the IS_REFINED_TO statement) and is what we believe is one of the important modeling concepts required in equational-based modeling.

A common mechanism for accomplishing deferred binding without explicitly using the IS_REFINED_TO statement occurs when we configure using merging (ARE_THE_SAME). The output stream from a reactor may be a liquid stream with the Unifac model defining its thermodynamic properties. We know the exit stream type within the context of the reactor as that is what the reactor produces. On the other hand, we have created the flash unit with the intention that it can accept any type of stream as an input. Thus we only require that it is a molar stream. If the liquid stream model is a refinement of molar_stream (as it is in our library defining stream types), then, when we effect the merge, the ASCEND system compiler will make the stream into a model of the more refined type before it does anything else in the process of merging.

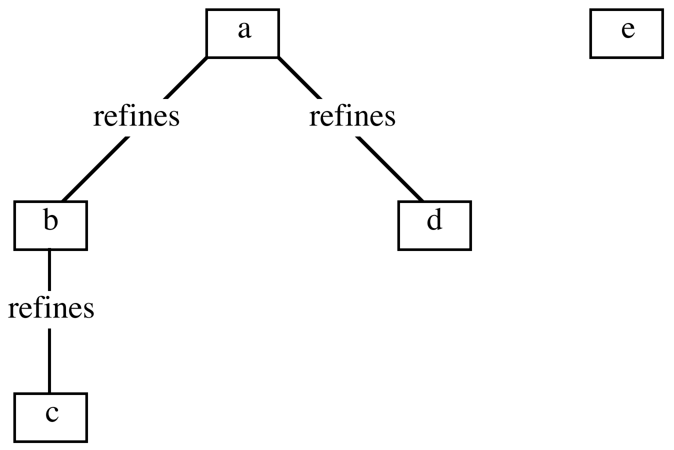


Figure 7. Diagram of the model hierarchy implied by code in Fig. 6

Running an ASCEND model through the interactive interface

Preliminaries

Definitions

Local machine	the computer sitting on your desk
ASCEND host machine	the computer on which ASCEND is running

Conventions in following text

Bold face	used for window names
Italics	used for button labels
`Courier font`	`used for instructions`

Buttons standards

Left mouse button	the normal button to use unless directed to use another one
Middle mouse button	often has no response when used. If used when opening a drop down window to show command options, allows you to keep the drop down window open and move it around on the screen.
Right mouse button	opens a help window for the object at which one is pointing

Remote login

The ASCEND system uses X-windows to display its windows. You can run ASCEND remotely from the machine on your desk provided it supports X-windows and is networked to the ASCEND host machine. The following commands are typical of those needed to log into the remote computer.

```
xhost +name_of_ascend_host_machine
telnet name_of_ascend_host_machine
setenv DISPLAY name_of_your_local_computer:0.0
```

The first command tells your local machine to accept X-windows input from the ASCEND host machine. The third command tells the ASCEND host machine to direct any X-windows output to your machine.

If the emacs (gnu-emacs) editor is available, the following command will start it up for you. You will need an editor to enter your own models into the system.

```
emacs&
```

The amphersand (&) causes the system to return control to you to run unix commands in your login window while starting up emacs in another window. If you have installed an ASCEND mode within the emacs editor, type

```
meta-X ascend-mode
meta-X abbrev-mode
```

while in the emacs editor to invoke it.

Setting up the ASCEND system

We solve an ASCEND model by starting up the ASCEND interface. We assume that someone has installed the ASCEND system already. A typical file system hierarchy for installing is as follows.

`/usr/local/ascend`	the main ASCEND subdirectory
`/usr/local/ascend/bin`	binary files
`/usr/local/ascend/lib`	compiled library objects
`/usr/local/ascend/man`	user man pages and help files
`/usr/local/ascend/help`	ASCEND help files
`/usr/local/ascend/include`	ASCEND source code
`/usr/local/ascend/TK`	source code for TK objects (objects used to create and drive the interface for ASCEND)
`/usr/local/ascend/models`	ASCEND model definition libraries
`/usr/local/ascend/models/README`	text document on model libraries (it is a good idea to read all README documents)
`/usr/local/ascend/models/libraries`	holds .lib files

`/usr/local/ascend/models/examples`	holds .asc and corresponding .s files for models one can run as is (without adding any further ASCEND model code)
`/usr/local/ascend/models/old`	contains old libraries and examples subdirectories
`/usr/local/ascend/model/pendings`	contains possible new or modified .lib, .asc and .s files which, if demand is there, will used to update the files in the current libraries and examples subdirectories

It is a good idea to create a file hierarchy within one's personal subdirectory that mimics a part of this structure. We typically have the following subdirectories set up for keeping our personal model definitions, where we keep the model definitions in the subdirectory asc and the scripts to run them in the subdirectory s. The subdirectory library is to keep any new library models one might be creating with the intent to add them to the system libraries later.

```
~/ascend/models/examples/asc
~/ascend/models/examples/s
~/ascend/models/libraries
```

It is very useful to add the following alias to your .chsrc file:

```
alias ascend /usr/local/ascend/bin/ascend3c
```

that will allow you to start the ASCEND system by simply typing in

```
ascend -c
```

when you wish it to start. The "-c" is an option that causes the system to give you access to some shell commands while running ASCEND[1]. For example, it does not support wild carding.

Running the ASCEND system

Start the ASCEND system. A number of windows will appear on your screen. The first, the small **Ascend IIIc** window, announces that one has started the ASCEND system. The animal icon in that window is the head of the animal known as a gnu, to remind you that ASCEND is distributed under a gnu licensing agreement.

Viewing the GNU license and warranty

To see the GNU license and warranty, click on the GNU window with the right mouse button to bring up the on-line help system. In the left window of the Ascend Help window, double click (left mouse button) on the entry GLOBAL/notices/GNU to display the subdirectories under that subdirectory. Then double click on the file entitled

```
GLOBAL/notices/GNU:GNU_License.
```

The warranty is

[1] The ASCEND system uses TK/Tcl for creating its interface. The shell commands are those TK allows.

```
GLOBAL/notices/GNU:GNU_warranty.
```

Setting the utility paths

Examine the window labeled **toolbox**. It has a first row of buttons labeled *utilities*, *help* and *exit* and a second row of buttons labeled *LIBRARY, SIMS, BROWSER, SOLVER, PROBE, UNITS, DISPLAY* and *SCRIPT*. These latter set of buttons identify the various tool kits available through the interface. It might be a good idea to open up each tool kit and move the window to a convenient location on your screen.

Click on the *utilities* button in the **toolbox** to open the ASCEND **Utility Settings window**. For the subdirectories as described above, the following are prototypes for the paths/commands you need to define for this tool. Type path names and commands on one line (even if we shown them on two or more lines here).

```
ASCENDIST dir           /usr/local/ascend
ASCENDHELP path         /usr/local/ascend/man/help
ASCENDLIBRARY path      /usr/local/ascend/models
                          /library/examples:
                        /usr/local/ascend/models
                          /library/libraries
Scratch directory       /usr/tmp
ASCENDUNITS dir         /usr/local/ascend/lib
Working Directory       ~/
Text edit command       /usr/local/bin/emacs19
Postscript viewer       /usr/local/bin/ghostview
Bug mail command        xterm-T Mailing_Bug_Report
                        -geometry+200+200 -e Mail -s
                        "ascend3c bug"
                        ascend+bugs@edrc.cmu.edu
```

Hit the **More** button to see the remaining settings.

```
Plot program name       /usr/local/bin/xgraph
Plot file type          xgraph   (plainplot and gnuplot are
                        also supported)
Text print command      lpr
                        (*if you would like to have the print
                        output show up in a file, use the
                        command
                        cat>>tmp.file
                        *)
PRINTER                 myprinter
Help mail command       xterm-T Ask_Experts_Mail -e Mail
                        ascend+bb@edrc.cmu.edu
TCL_LIBRARY             /usr/local/ascend/lib
TK_LIBRARY dir          /usr/local/ascend/lib
Font Selector           /usr/ucb/bin/xfontsel
Spreadsheet command     (* your choice here *)
```

After entering these in, be sure to hit the **Save** button to save them if you wish them saved.

Example 1: Running the simple flowsheet model

We shall solve the simple flowsheet example that is defined in Figs. 4, 5 and 6. We assume you have started the ASCEND system as described above.

Running a script

- (**Optional**. In the emacs editor, open the file you are about to use by executing an instruction similar to the following (modify this as needed to account for where the examples subdirectory is actually stored in your ASCEND installation):

  ```
  ctrl-x ctrl-f /usr/local/ascend/models/examples/simple_fs.asc
  ```

  The editor will now contain the file of ASCEND code that defines the simple flowsheet model we discussed earlier in this document. You will not be able to change the contents of this file as it will be read only. Browse this model.)

- Click on the button labeled **SCRIPT** in the toolbox to open the **SCRIPT** tool kit (if it is not already opened).

- Open the file "simple_fs.s" within the **SCRIPT** window by mousing the *Edit* button and picking the *Read file* command. A **File select box** will open. The file "simple_fs.s" is located in the **examples** subdirectory. You may wish to resize the script font size if it is hard to read.

- Lines preceded by a pound sign (#) are comments. Skip past them and highlight the instruction starting from the

  ```
  set library
  ```

  instruction through to the end of the

  ```
  READ $examples/simple_fs.asc;
  ```

  instruction by "dragging" the mouse over them. Execute the highlighted instructions by clicking on *Execute* in the header of the **SCRIPT** tool kit and then on *Statements selected*. ASCEND loads the libraries indicated in the highlighted instructions and then the model (simple_fs.asc) into the **LIBRARY** tool kit.

  (Hint: If you wish to highlight a single instruction in the **SCRIPT** took kit, triple click anywhere in that instruction.)

- Continue with the next instruction in the **SCRIPT** (i.e., highlight it and then mouse on *Statements selected* under the *Execute* button) and observe what happens as it executes . ASCEND will compile an instance of *test_controller*, calling it *tc*.

- Select *tc* in the **SIMULATIONS** tool kit, click on *Export* and then *to Browser*. You have just exported the compiled model to the **BROWSER** where you can look at it.

- At the bottom of the **BROWSER** window is a button labeled Atom Values. Clicking on it toggles whether the system will display variable values or variable type. Click on it a two or three times while watching the entry for `conv` in the RHS window change between

  ```
  conv = 0.5
  ```

  and

  ```
  conv IS_A fraction
  ```

  Turn this button on as it is usually more useful to display the values for variables. The value of 0.5 is the system default value for this variable. (We have found zero to be a dangerous default value.)

- Run the next instruction in the **SCRIPT** which runs the values procedure in tc. To display the code for this procedure, return to the **LIBRARY** tool kit, pick the model *test_controller*, and under the *Display* tool set pick *Code* (i.e., click in turn on *LIBRARY* in the toolbox, on *test_controller* in the right window in the **LIBRARY** tool kit, on *Display* in the tool sets listed at the top of the **LIBRARY** tool kit and finally on *Code* in the window that drops down).

  You can prove to yourself that the system has run the values procedure. First note that the value for conv is now `0.07`. Is that the value it should be? The real question is: Can you get the code for the `values` procedure to display? Note first that `tc IS_A test_controller` (look in the **SIMULATIONS** window). Pick in turn *test_controller*, *Display* and, *Code* in the **LIBRARY** tool kit. In the procedure `values`, `conv` is indeed set to `0.07`. The `values` procedure executes the procedure `fs.values` to set many other values for this example problem. Looking above in the code, note that `fs IS_A test_flowsheet.` You need to display the code for `test_flowsheet` to see its `values` procedure. Can you do that?) Almost the entire model for `test_flowsheet` is the `values` procedure for it. Note that the contents of the **DISPLAY** can be scrolled. For convenience, you can also enlarge the **DISPLAY** window and move it to a more convenient position.

- Pick any item listed in the right hand window of the **BROWSER** - e.g., *fs*. Note ASCEND places it into the LHS window and makes it the current object. Continue exploring the entire model. Does it make sense to you? Check other values that were to be initialized. Were they?

- Prove that the model is "wired" up correctly. Pick *tc* as the current object by clicking on it in the LHS window. Next pick *fs*, then *m1*, then *feed*, then *[2]*. The current object is now tc.fs.m1.feed[2]. Under *Find* (third tool set at top of **BROWSER**), pick *Aliases*. The window that opens displays two different names. Both these names are for the current object. The first says that the second feed to the mixer m1 is also the second output stream from the splitter sp1. Is this right?

Try showing that the entire flowsheet is correctly wired up by looking at all the input and output streams to all the units in tc.fs.

- To see how the values procedure is picked through the interface, make tc the current object. Then under *Edit* pick *Initialize*. A window opens listing the available procedures you can pick to run. If you do not see a values procedure, did you make tc the current object? Pick *values* and *OK* (or *Cancel* as the values procedure already executed).

- Make the current object *tc*.

- Under *Export*, pick *to SOLVER*. The **SOLVER** window opens (move it to a convenient spot) along with a window labeled **Eligible** that lists a number of variables. Can you decide why this window might have opened? Remember, ASCEND has tools to tell you of errors you might be making. Hit *Cancel* for this window. We will return to this issue momentarily.

  Read the information displayed at the bottom of the **SOLVER** window. This model has 43 relations, of which 43 are equalities which are included and none are inequalities. It has 55 variables and all are mentioned in at least one of the 43 included relations (i.e., there are no unattached variables). All 55 of the variables are free to be computed (their fixed flags are equal to FALSE).

  To see a fixed flag, return to the **BROWSER** and pick *tc.conv* to be the current object. On the RHS will be its parts including one called *fixed* which has a value FALSE.

  Returning to the **SOLVER** tool kit, the last information entry says this model is *under* specified by *12*. It is telling you that you need to fix (specify) 12 variables before the problem is a well-posed one comprising 43 equations in 43 unknowns.

  Now can you figure out why the **Eligible** window dropped down when you sent this model to the **SOLVER**? Yes, it was telling you that you need to fix one of the variables listed to start the process of fixing the 12 variables needed to make this problem well posed.

  To get the **Eligible** window to reappear, You will need either (1) to return to the **BROWSER** tool set and re-export to the **SOLVER** or (2) use the **SOLVER**'s *Import* button to restart the degrees of freedom analysis.

  Pick *tc.conv*. The **Eligible** window disappears for a split second and reappears asking for you to fix another variable. Hit *Cancel* at this point as this is not the way you need to proceed to pick variables to fix. Note, the model is now underspecifed by 11 variables.

- Run the *reset* procedure for tc (either by returning to the **SCRIPT** or by returning to the **BROWSER** and finding and executing it). Run it from the **SCRIPT** to be sure you make no errors this time. The numbers change immediately in the **SOLVER** window, indicating now that 12 of the variables are now fixed, leaving 43 which are free. The system is square - it has 43 equations in 43 variables.

- Solve the model by running the next instruction in the **SCRIPT**. Alternatively, you can pick the *Execute* tool set in the **SOLVER** and under that the *Solve* tool.

- Explore the model in the **BROWSER**, examining the values for all the variables.

Changing what is fixed and resolving

At this point you are ready to have some real fun with ASCEND. You can reset values for some of the fixed variables and resolve. You can change the status of a variable from fixed to free or vice versa and solve again. The following instructions will lead you through doing this latter type of analysis.

- Make tc.fs.fl1.liq.f['c'] the current object in the **BROWSER**.

- Note that its fixed attribute is FALSE (RHS window in the **BROWSER**). That means ASCEND computed it. Let's try fixing this variable and then altering its value and resolving.

  Double click on the fixed attribute (in the RHS window) using the middle button. The value for fixed will toggle to TRUE. Because the model was well-posed already, ASCEND drops a window called **Overspecifed** listing a number of currently fixed variables, one of which you need to unfix to make the model well-posed again. Enlarge this window or scroll it to see all the options. Pick *tc.conv* and then *OK* in that window. Note that ASCEND has altered the fixed flag for tc.fs.fl1.liq.f['c'] to TRUE.

- Under the *Edit* tool set, pick *Set value*. The value 431.938 should be displayed. Edit it to a nearby value, say 420.0. Make tc.fs.fl1.liq.f the current object and verify that the value of ['C'] is 420 mole/s.

- Resolve the model. The model will very quickly resolve.

Using the PROBE

By now it should be evident that you would like to monitor some of the variable values while changing others. You can use the **PROBE** tool set to aid in this monitoring. Try the following.

- Make the current variable in the **BROWSER**, tc.fs.fl1.liq.f['c'] and *export* it to the **PROBE** (Use the instruction *Item to Probe* under the *Export* tool set in the **BROWSER**). Also export tc.conv. (As time progresses, you can change which variables are in the **PROBE**. Note there are five different **PROBE** collections which you can construct for display.)

- Pick the first variable displayed in the **PROBE** (yes, click on it). Export it to the **BROWSER** where it now becomes the current object.

- Alter its current value to 400.0 (*Edit*, *Set value*). Note it immediately has this value by watching the **PROBE**.

- Move the **BROWSER**, **SOLVER** and **PROBE** tool sets (you may want to resize the **PROBE**) so they do not overlap each other on the screen. You will want to watch all three.

- Note the value of tc.conv and then solve the model. Note how tc.conv changes from 0.0672556 to 0.0628319.

Clearing and starting again

Okay, now you can do a lot of playing with this model. Try anything you would like to do. You cannot hurt the model nor the ASCEND system. If all else fails, you can start again by doing the following.

- Open *SIMS* (the **SIMULATION** tool kit).

- Select tc and, under the *Edit* tool set, *Delete* it.

- Return to the **SCRIPT** and run all the instructions starting with the *COMPILE* instruction .

  Alternatively you can recompile it yourself by opening the **LIBRARY** and highlighting `test_controller`. Under *Create*, pick *Compile*. Enter the name `tc` (unless it is the name the system offers to you already). Hit *OK*. The **SIMULATION** tool kit opens with the compiled object tc. Export to the **BROWSER** and, if you like, to the **SOLVER**.

Putting variables values in the spreadsheet

ASCEND will be able to put values into a spreadsheet so you can see them all and manipulate them. This option is not yet implemented.

Debugging tools

There is a debugging tool in ASCEND which allows you to examine the solving activity in more detail.

<u>Altering SOLVER attributes</u>

- Click on the radiator like icon next to the word SOLVER. You will see a list of solvers. Pick the one called *Slv*. The system opens a window displaying for you an list of attributes for the Slv solver. You can alter any of these by simply editing it. For example, altering the detailed solving info. required attribute will cause ASCEND to output lots of output into the window in which you started the ASCEND system.

<u>Viewing the incidence matrix for the model</u>

- In the **SOLVER** tool kit, under the *Display* tool set, pick the tool *Incidence matrix*. The system opens up a window labeled **INCIDENCE** and displays a matrix containing dark squares and possibly some x's. This matrix has one row for each equation in the model and one column for each variable. The squares and x's indicate that an equation contains an explicit mention of that variable in it.

The system shows incidences for unfixed variables as squares and x's for fixed.

To get us back to a standard state, return to the **BROWSER** and make `tc` the current object. Under the *Edit* tool set, pick *Initialize* and run the *reset* procedure. This procedure fixes 12 variables and makes the problem square. The first 43 columns of the incidence matrix will now contain squares and the last 12 x's. Mouse on any position you wish in the incidence matrix. The corresponding equation and variable internal numbers and names and block number show up in the fields at the top of the window.

ASCEND partitions and precedence orders the equations and variables when solving them. It does this step as soon as the problem is square -- so it is already done here. A partition is a block of k ($k \geq 1$) equations in k variables which has to be solved simultaneously when solving the problem. A precedence order is the order in which the system can solve the blocks, one after the other, to solve the entire problem.

Start at the upper left and mouse the diagonal incidences one after the other. The first equation and variable are in block 0. The next 30 equations and variables are in block 1. The next four are in block 2, and so forth. ASCEND will solve the first equations first and then the first 30 equations simultaneously for the 30 new variables they introduce to the problem. Then it will solve the next four equations for the four new variables these equations introduce, and so on.

There is a better way to find the blocks than this which we shall examine next. However, it is often very useful to see a display of the structure of the equations as shown here. Note, you can discover the fixed variables quite easily by mousing on the columns containing an x.

Running the debugger tool

- In the **SOLVER**, under *Analyze*, pick *Debugger*. You can now pick a variable, an equation, or a block of equations by number and examine it. For example, enter the integer 0 (zero) in the window just below the word *Variable:* and hit the button *Name* just below this value. The message

```
tc.fs.sp1.split[2] 0.99
```

will appear in the **xterm** window from which you started the ASCEND system (way back where you typed "ascend -c" to start ASCEND). This window is the standard output for ASCEND where the compiler puts out it messages, and so forth. This message about variable 0 is telling you the name and value of the variable that ASCEND has called variable number 0. (Numbering of variables and equations starts with 0 in the **Debugger**.) Hitting the *Attributes* button gives no added information for variables. The *Export* button allows you to export this variable to the **BROWSER** or the **PROBE**.

What is variable 10?

Try finding all you can about the third equation. Note, it too can be exported to the **BROWSER** and **PROBE**.

Now find out about block 0 (1 equation and variable) and then block 1 (30 equations and variables).

Let's put the variables for this block into the **PROBE**. First open the **PROBE** and open the second probe window (click on 2 at the bottom). This window should be empty. Return to the **Debugger** and hit the *Export to probe* button under the *Block*: options. The first 30 entries is a list of the equations in this block. The last 30 are the variables and their values.

You can also get the **Debugger** to show you the residuals for the equations in this block. Hit the *Equations* button and then the *Residuals* button in the drop down window that appears. The system lists the residuals in the standard output window (the xterm mentioned above). If you are trying to solve a hard model and it will not converge, the **Debugger** allows you to find which equations are not converging.

Finally at the bottom you can do things to all the variables and equations in the system.

Explore the features of the **Debugger** by trying them.

Find dependent equations

In the **SOLVER** under analyze is the tool to find dependent equations. *Find dependent eqns*. looks for equations that are causing the problem to be structurally or numerically singular. When checking for numerical singularity, it examines the linearized equations the solver Slv solves at each Newton iteration. If it finds an equation it cannot pivot (all alternative pivots in it are too small), it lists the equation and tells you the linear combination of the other linearized equations which give this one.

It only tells you the equation numbers so you must use the debugger tool (see just above) to discover which equations are involved. If the equations are nonlinear, this dependence is only local. One only has the first clue of a problem. Try altering the values given to the fixed variables in such a way that the system computes different values for the variables in the equations involved in this linear dependence. If the problem persists and the exact same equations are still dependent, have your second clue that you have a nonlinearly dependent equations in your model. Next alter what is fixed and what is unfixed among some of the variables in these equations. If the problem still persists among the exact same equations, look for a dependent equation. It is particularly easy to spot it the system reports only a few equations all with small integer coefficients in the dependency is it finding. For example, it might continually tell you that equation 5 is -1* equation 3.

Other *Analyze* tools

Under the Analyze tool set in the **SOLVER** are three other tools that can aid in debugging. *Over-specified* analyzes the **SOLVER** object and produces a list of variables, one of which you must release for the system to be square. If more than one has to be released, you must release one by picking one on the list and then reanalyze using *Over-specified* again.

Evaluate unincluded eqns. evaluates the residuals for all the equations which were not included while solving and reports those which are not satisfied.

<u>Other tools to become available</u>

Many other tools will exist in ASCEND. Their functionality is suggested by their names. If not implemented, they will always be grayed out as buttons when you look at them. We welcome suggestions for new tools. Development in concert with users is one of the hallmarks of the ASCEND system.

Example 2: Executing a library model

We shall now try a much more difficult task using ASCEND: picking a model in the ASCEND library and testing it. The model is simple_column found in the file (assuming the file system structure given earlier):

```
/usr/local/ascend/models/libraries/column.lib
```

Open up this model using the editor so you can explore it.

We proceed as follows.

- First we will need to create a test model for the model *simple_column*. We open a new file in the editor, calling it "**test_simple_column.asc**" in which we shall construct this model.

There is no documentation to read about this model. All our information about the model is in the ASCEND code itself. So this exercise is a test of our ability to read this code and make sense of it. Looking at the code we first note that the *simple_column* model is a refinement of a *tray_stack* model. We see that the *tray_stack* model is in the same library; we should also look at it.

In *simple_column*, we see a variable called *feed_loc* which is of type *integer*. We guess this to mean the tray number onto which we place the feed. The second statement seems to be defining slack variables for an overmaterial balance, one per component.

The next few statements refer to the trays, starting with tray[1] which is refined to be a *condenser*. It appears we are numbering from the top of the column down. The tray pointed to by *feed_loc* is refined to be a *simple_feed_tray*, supporting our earlier conjecture. The tray pointed to by *ntrays* is refined to be a *reboiler*, and all other are set to be of type *simple_tray*. So a simple column has a single feed tray. We guess that it has products withdrawn from only the condenser and reboiler as the rest are simple trays. Overall, that seems logical.

Who defined the variable *ntrays*? It is not defined in this model so it must be a part of the model tray_stack. We can check and discover that it is. We also note the existence of the set called *components*. It is of type symbol. It must be a list of the components in the column.

We can now start to construct the test routine. We place the following statements into the file **test_simple_column.asc**. The procedures are there because we are following the good style this document strongly advocates.

```
MODEL test_simple_column;

   c                                      IS_A  simple_column;

   c.ntrays                               := 10;
   c.feed_loc                             := 5;

   c.components                           := ['A','B','C'];

   INITIALIZATION

      PROCEDURE clear;
         RUN c.clear;
      END clear;

      PROCEDURE specify;
         RUN c.specify;
      END specify;

      PROCEDURE reset;
         RUN clear;
         RUN specify;
      END reset:

      PROCEDURE values;
      END values;

      PROCEDURE scale;
         RUN c.scale;
      END scale;

END test_simple_column;
```

Writing the values procedure

Remember that the procedure called values has to provide values for all the fixed variables for the model. We have not finished it as we do not know which variables the author of this model (Bob Huss) decided to fix. We could guess, but that looks hopeless (and it is). We could also try to trace recursively through the specify procedures.

Trying that, we see that the *simple_column* specify procedure first runs its own *seqmod* procedure. Okay, so we switch our attention to that procedure. It first runs the *seqmod* procedures for all the trays. One by one we need to examine the models for the trays. The top tray is a condenser. Where is its model?

We do not find it in this library, but, at the top of this library, Bob has listed all the libraries this library needs. One is called *flash.lib*. Looking in there we find the *condenser* definition. (Our sanity is beginning to slip and our stomach is churning. This approach is not looking very promising, but we persist.)

We locate the *condenser* model (we see it refines the *VLE_flash* model). We find its *seqmod* procedure. The first thing this procedure does is run the *specify* procedure for VLE. Huh? What is VLE? It is not in this model. Oh, yes, it could be in the *VLE_flash* model definition. At this time, we look around the room and suggest that it is time for coffee.

Of course, we headed down this path here only to show how hopeless it is. There must be a better way.

Finding things by type

Why not let ASCEND tell us which variables Bob has fixed? Can it? There is a tool in the *Find* tool set within the **BROWSER** tool kit that is called *By type*. In other words, the name suggests we can find things by type. To find by type means that we can locate any part within the compiled model by describing its attributes. We want to find all the variables (all are of base type solver_var) in the compiled model which have their fixed flag set to TRUE.

Proceeding as follows, we shall now use ASCEND to help us write the values procedure for our test model.

- Start the ASCEND system (if already in the ASCEND system, return to the **LIBRARY** tool kit and run the instruction *Delete all types* under the *Edit* tool set.)

- Open the **SCRIPT** tool kit and set it to record instructions (push the button on the bottom so it read Record On.

- Next return to the **LIBRARY** tool kit and load the following libraries:

  system.lib
  atoms.lib
  plot.lib
  components.lib,
  H_G_thermodynamics.lib
  stream.lib
  flash.lib
  column.lib

  (Note, these are listed at the beginning of the **column.lib** description as being required for using the **column.lib**.) If you have been watching the **SCRIPT** tool set, you will note that the system is writing a script for you which will allow you to repeat all these steps using it later.

- Load our newly created file "**test_simple_column.asc**."

- Next compile **test_simple_column**, calling the compiled object "tsc."

- In the tool kit **SIMULATIONS**, select tsc and export it to the **BROWSER**.

- In the **BROWSER** under the *Edit* tool set, select *Initialize*. Select *reset* and then *OK*. The column specify routine has now set the fixed flag for all the variables that Bob fixed to make his column model well posed.

- Export tsc to the **SOLVER**. It has 303 included equalities and 349 variables. One of the variables is unattached, meaning it is not used in any of the equations. That is not a problem; the solver will ignore it. The *reset* procedure has set the fixed flag for 45 variables to TRUE. The **SOLVER** reports the model is square. The system has found it can use the 303 equations to solve for the remaining 349 - 1 - 45 = 303 variables. This discovery is based only on the structure of the equations (i.e., the equations have a legal transversal (output set assignment)). They may, however, be numerically singular, something not detected by this preanalysis.

- At this point, turn off the record instruction in the **SCRIPT** tool kit.

- Go to the **PROBE** and pick an empty window (or clear the current one).

- Go back to the **BROWSER** and run the tool *By type* in the *Find* tool set. You are going to use this tool to have the system find for you the names of 45 variables that Bob has fixed with his version of the *reset* procedure.

  The **Find by type** window will open. It has four user input windows in it, into which you enter the type of item you wish the system to find in the first window. For example, if you wished to find all items of type VLE_flash (all trays are of this base type), enter VLE_flash and ask it to do the search. If the type is not a `solver_var`, the other windows should be empty.

  If the type is `solver_var`, you can further screen which items of that type you wish to find. The system bases the search on the attribute[2] whose name you enter in the second window and whose value you enter into the third window. Make sure the fourth window is empty unless you wish the search to be over a range of numerical values when the attribute of type REAL. In this case the third window contains the lower bound for the range and the fourth the upper bound. An empty fourth window indicates that the search is for the value in the third window; a nonempty fourth window means the search is to be over a range of values.

  The windows have default values in them of `solver_var`, `fixed`, TRUE and empty, respectively. Accept the default values as they pose exactly the search you wish to execute; i.e., find all items of base type `solver_var` whose fixed flag is set to TRUE.

- Export all the variables that the system found to the **PROBE**.

The **PROBE** contains all the variables that Bob fixed for his model. The instruction

```
   tsc.c.trays[1 ].alpha['A'] = 1.00000
```

[2] Note that one special attribute is the value of any REAL (solver_var is a refinement of REAL). It is known as VALUE (all capital letters).

shows up for all trays and for all species (it is a good idea to double check that it shows up for ALL trays and ALL species). We can compactly add instructions in the **values** procedure of our test model to set the relative volatilities for all the species on all the trays. The instructions:

```
   c.trays[1..c.ntrays].alpha['A'] := 2.0;
   c.trays[1..c.ntrays].alpha['B'] := 1.5;
   c.trays[1..c.ntrays].alpha['C'] := 1.0;
```

do that. Note, that the tsc part of the instruction is removed as we are adding this instruction into our test model of which tsc is a compiled instance. Also, note that we must refer to *ntrays* as *c.ntrays*. Not labeling an item with its part name, here "c," is perhaps the most common error one makes when programming in ASCEND.

As we account for setting the value for each variable, it is convenient to delete it from the **PROBE**. We delete a variable from the **PROBE** by picking the variable and using the *Remove selection* instruction under the *Edit* tool set. We shall continue until we have looked at all variables listed in the **PROBE**.

The instruction

```
   tsc.c.trays[6].cmo_ratio =  1.00000
```

shows up for only trays 2, 3, 4, 6, 7, 8 and 9. Bob apparently has set it for all the simple_trays. We need to know what this variable is before we can decide how to set it. (One way is to call up Bob Huss and ask him (412-268-5212). Another is to play detective on the models and try to find where it is defined. Returning to the document listing the libraries, we look for the model **simple_tray.** It is located in **flash.lib**. This model defines *cmo_ratio* as a factor (a dimensionless real) and uses it in what looks like a material balance. If we set *cmo_ratio* to one, the material balance says the total liquid flow into the tray equals the total liquid flow leaving. Ah, cmo must stand for constant molar overflow!

So why is this variable there? Why not just write that the total liquid molar flow into the tray equals the total liquid molar flow leaving? Perhaps ours is not to understand. On the other hand, we know that the use of a constant molar overflow assumption replaces a heat balance for a stage. A plausible answer is that Bob wants his simple_column model to be more general. For example, if he adds a heat balance to the equations defining the tray, then the liquid flow in would not generally equal that out. To allow this to happen, he could add in the heat balance (taking away a degree of freedom for the model) and then give it back by making *cmo_ratio* a computed variable.

We set the *cmo_ratio* for all the *simple_trays* to 1.0.

```
   c.trays[2..c.feed_loc-1].cmo_ratio           := 1.0;
   c.trays[c.feed_loc+1..ntrays-1].cmo_ratio    := 1.0;
```

tsc.c.reduce is more of a mystery. Bob really should add a comment on what it is. (Shame on him, right?) On asking him, he uses it to aid in numerically closing the heat balance for a tray when one is included in later refinements of the tray models. Supposedly it can be set to any value desired here.

```
   c.reduce := 1.0;
```

We notice next that the *reflux_ratio* is fixed. We see that it is a variable within the first tray which we know is a *simple_condenser*. Looking at the *simple_condenser* model in **flash.lib**, we see that *reflux_ratio* has the standard definition. The reflux in the column (the stream liquid['liquid']) has a total flow equal to the *reflux_ratio* time the distillate top product flow. When looking at that equation, we note it mentions a possible vapor product.

In the list of fixed variables in the **PROBE**, we see next one called

```
   tsc.c.trays[1].VLE.phi['vapor'].
```

Is this somehow related to the permitted vapor product at the top? Believing it is, we set it to zero (when the model solves, we must check out this conjecture).

```
   c.trays[1].VLE.phi['vapor'] := 0.0;
```

We see that the distillate total liquid flow out is fixed. This makes sense. It is a common specification for a column. We see also that the feed flows for each of the species are all fixed. Again, that makes sense. We add statements to specify them using a rule of thumb noted earlier that a normal sized chemical process has a feed flow about 1 kg/s. A typical molecular rate is 100 kg/kg_mole so we can have a feed about the normal size if we use the values for the flow of about 0.01 {kg_mole/s}.

```
   c.trays[feed_loc].input['feed'].f['A'] := 0.01 {kg_mole/s};
   c.trays[feed_loc].input['feed'].f['B'] := 0.01 {kg_mole/s};
   c.trays[feed_loc].input['feed'].f['C'] := 0.01 {kg_mole/s};
```

One could also let ASCEND do these computations for you by entering the values as:

```
   1.0 {kg/s*kg_mole/100.0/kg}
```

We would like our column to produce A at the top and B and C at the bottom, so we can now choose a value for the distillate top flow. To separate A from B and C, one would set the distillate top flow to the molar flow of A in the feed. The distillate will not be pure as the column is not infinite in size nor will we use an infinite reflux ratio, but this value is the best setting for its flow.

```
   c.trays[1].liqout['distillate'].Ftot :=
            c.trays[feed_loc].input['feed'].f['A'];
```

Next we spot a variable called *q* which is set for tray 5. Textbooks typically call the thermal quality of the feed to a column *q*, where *q* is the fraction of the feed which joins the liquid stream going down the column; *1-q* is the fraction that joins the vapor stream. We look around for the definition of *q*. It is referenced as

```
   tsc.c.trays[5].q
```

which makes it a variable belonging to the type *simple_feed_tray* (which is the type for tray[5]). Locating *simple_feed_tray* in **flash.lib**, we see it introduces *q* and uses it in a material balance exactly as we expected. We set *q* to 1.0 so all the feed will be treated as saturated liquid.

The final variable we need to specify has the name

```
   tsc.c.trays[10].vapsplit['vapor_product']
```

which will again require some educated guessing. It seems to be the vapor split for the vapor product in the reboiler. We look at the reboiler model in **flash.lib**. It is not there. The *reboiler* model refines the *VLE_flash* model; we look at it (again, in the **flash.lib**). We find the variable about a third of the way into the model definition (somewhat before the *INITIALIZATION* section). It is defined as an array over the set *vapout.* A few statements later it is used in an equation. Close examination of the equation suggests it is a material balance. It seems to be the fraction of the vapor produced in the flash that exits in a particular vapor output stream. The stream is called *vapor_product* here; we want no vapor product from the reboiler, so we set this value to zero.

The meanings for some of these variables are at best conjectures on our part at this time. We will take our chances that we have them right. The final *values* section for our model becomes:

```
   PROCEDURE values;

      c.reduce                                  := 1.0;
      c.tray[1].reflux_ratio                    := 2.0;
      c.tray[1].VLE.phi['vapor']                := 0.0;

      c.tray[2..c.feed_loc-1].cmo_ratio         := 1.0;
      c.tray[c.feed_loc+1..c.ntrays-1].cmo_ratio:= 1.0;

      c.tray[1..c.ntrays].alpha['A']            := 2.0;
      c.tray[1..c.ntrays].alpha['B']            := 1.5;
      c.tray[1..c.ntrays].alpha['C']            := 1.0;

      c.tray[c.feed_loc].q                      := 1.0;
      c.tray[c.feed_loc].input['feed'].f['A']   := 0.01 {kg_mole/s};
      c.tray[c.feed_loc].input['feed'].f['B']   := 0.01 {kg_mole/s};
      c.tray[c.feed_loc].input['feed'].f['C']   := 0.01 {kg_mole/s};

      c.tray[1].liqout['distillate'].Ftot :=
            c.tray[c.feed_loc].input['feed'].f['A'];

      c.tray[c.ntrays].vapsplit['vapor_product']:= 0.0;

   END values;
```

(This version is the result of attempting to run it and then removing the syntax errors from it. Even with experience, we made all sorts of errors (like calling it *feed_loc* instead of *c.feed_loc*).)

- Using the editor, add these statements to the definition of our model *test_simple_column* in the file **test_simple_column.asc**.

Solving the model

- In the tool kit **SIMULATIONS**, delete *tsc*.

- Return to the **SCRIPT**, pick the last three instructions and execute them (READ FILE test_simple_column.asc, COMPILE tsc, and RUN {tsc.reset}). The READ FILE instruction will replace the previous file of the same name IF the time stamp on the file to be read is newer than the time stamp of the one already there, else the system ignores this instruction. Since you have edited and replaced this file since last reading it in, it gets replaced. You can of course display this file from the **LIBRARY** if you wish to be certain which version is now there.

- Start recording in the **SCRIPT** again.

- Return to the **BROWSER** and run the *values* procedure for the model. Export *tsc* to the **SOLVER** and solve. If all was done correctly, it will solve quickly (ours did when we tested it).

Plotting the values

It would be very useful to plot the values that this column model has produced. Look again into the file column.lib using the editor. You will find a model called plot_column which is just before the simple_column model. It contains the part

```
col                                    IS_A tray_stack;
```

Now we shall see the power of defining models using a refinement hierarchy. A major purpose of defining one model as a refinement of another is that the system then knows that the more specialized model can replace the less specialized one in any model in which it exists. The `simple_column` model refines `tray_stack` also, and your column, `c` in the above, is a `simple_column`. It is a specialization of a `tray_stack`, therefore.

Suppose we include in your test_simple_column model a part we call `plotcol` and declare it to be a `plot_column`:

```
plotcol                                IS_A plot_column;
```

We next add the statement

```
plotcol.col, c                         ARE_THE_SAME;
```

What has happened here? Is this legal? Yes, it is because `plotcol.col` and `c` are both of type `tray_stack`, the latter being a refinement of `tray_stack`.

Inside the plotcol model are two items of type plt_plot_symbol. The system can plot any item of this type. To get a plot proceed as follows:

- Edit in the above two statements into the `test_simple_column .asc` model definition. They should follow the statement defining `c` as a simple column.

- In the **SIMULATION** tool kit, delete *tsc*.

- Return to the **SCRIPT** and run the instructions that read in and compile `test_simple_column.asc`, the instruction that run reset and values for it, and finally the instruction that solves it.

- Return to the **BROWSER** and make *tsc.plotcol* the current object. Under *Edit*, run the *Initialize* tool and select *plot_values*. Bob's procedure for this maps values from the column model into the variables we are about to plot.

- Make the current object in the **BROWSER** *tsc.plotcol.plotx*. Under **DISPLAY**, select *Plot* (which is now allowed as the current object is one the system knows it can plot). A plot will appear on your screen showing the liquid composition up and down the column. Remember that trays are numbered from the top so the top of the column is to the left on this plot.

- Repeat the last instruction but make *tsc.plotcol.ploty* the current object to see the vapor compositions.

Checking conjectures on the meanings of the variables

Now would be a good time to verify all the conjectures about the meaning of the variables. We can check the following by browsing the model.

1. Is there a vapor product from the condenser and, if so, does it have a zero flow rate as we wanted?

2. Is there a vapor product from the reboiler and, if so, does it also have a zero flow as we wanted?

3. Does the liquid flowing down the column increase in flow rate as it passes by the feed tray by an amount equal to the feed flow rate (0.03 {kg_mole/s})? It should.

4. Does altering the value of the variable *tsc.c.reduce* seem to alter any of the solution values? It is not supposed to if our understanding is correct.

5. Is the column wired up as expected?

Converting the column model to a rigorous one interactively

We can interactively modify the types of the parts included in this model, incrementally converting it into a rigorous column model which uses nonideal thermodynamic models to evaluate the physical properties of the three species. To accomplish this latter conversion, we have to associate the species with actual ones in the **components.lib**. The interactive incremental compile statements are included in the **BROWSER** under the *Edit* tool set.

Exploring the buttons

You should explore all the buttons in the interface methodically. If you do the exploring with the small sample problems above, you can do little harm to anything even if you goof.

Note, if the core dumps, be sure to issue the instruction

```
rm core
```

to get rid of the very large file it has created (unless you know how to use a core dump to diagnose a problem).