This memo explains how to access ABACUSS II (Advanced Batch And Continuous Unsteady State Simulator). ABACUSS II will be used to simulate the distillation tasks and the first reaction task during the design of the Lucretex monomer process. You will be using ABACUSS II’s dynamic simulation features. Dynamic simulation is the use of mathematical models to predict the time dependent or dynamic behavior of a chemical process (as opposed to steady-state simulation which is only concerned with predicting the steady-state of a process).

Accessing ABACUSS II on Athena

At the moment ABACUSS II is available on Athena only for Linux platforms (denoted PC/LINUX on the Athena cluster map). To run the ABACUSS software, you first need to setup the file systems containing the program. Please add the course locker by typing

\texttt{athena\% add 10.490}

at the Athena prompt. You will need to add the course locker each time you want to call ABACUSS or to access the contents of the locker. The first time you want to use ABACUSS type:

\texttt{athena\% setup_abacussII}

This command creates a directory called \texttt{ABACUSSII} in your home directory. You will notice that three subdirectories have also been created: \texttt{input}, \texttt{include} and \texttt{output}. All input files for ABACUSS should be placed in the \texttt{input} directory with the exception of files that are included within other files and are placed in the \texttt{include} directory. Results computed during ABACUSS simulations are by default written to the \texttt{output} directory (see below for overriding this default).

If you look in the \texttt{input} and \texttt{include} directories, you will see that a number of ABACUSS input files to be used during the Lucretex project have been copied over automatically from the 10.490 course locker. All ABACUSS input files are suffixed .ABACUSS. If you need fresh copies of these files, they may be copied from the 10.490 locker as follows:

\texttt{athena\% cp /mit/10.490/Public/ABACUSS/Series_Reactions.ABACUSS \textasciitilde/ABACUSSII/input/}

This copies the input file \texttt{Series_Reactions.ABACUSS} from the course locker into your input directory.

The command \texttt{setup_abacussII} copies the input files from the course locker to your \texttt{input} directory only if your input directory does not contain them, so any changes you make to the files will not be lost. However, before editing any of these files we recommend that you make a copy, and edit the copy only, leaving the original intact for reference. For example:
To start up the ABACUSS II modeling environment, type the following at the Athena prompt:

```
  athena% cd ABACUSSII
  athena% abacussII &
```

These commands change the current directory to ABACUSSII, sets up the correct library path for use of ABACUSS and calls the modeling environment. It is important to always run ABACUSS in the ABACUSSII directory because the plotting environment looks for an output subdirectory in order to plot the results of simulations. The ABACUSS graphical user interface (GUI) should be popped up onto your screen. The GUI has a default text editor that has syntax highlighting for the ABACUSS II modeling language, but if you prefer you can use another editor. There are various buttons on the GUI which may be selected by clicking on the left mouse button. This document attempts to explain how this program can be used to develop and run a process simulation. In addition to providing a text editor, the GUI is used to manage the files that you are working with, and provide a means of selecting different options. You may exit ABACUSS by selecting the **Exit** from the **File** menu.

From now on, when you log onto Athena and want to run ABACUSS type:

```
  athena% add 10.490
  athena% cd ABACUSSII
```

Figure 1: ABACUSS Graphical User Interface
i.e., you don’t need to type setup_abacussII again.

Tutorial Example: Simple Mass Action Kinetics

This section guides you through a simple tutorial introduction to the ABACUSS input language, and how to run an ABACUSS simulation. We consider the following system of sequential irreversible elementary reactions:

\[ A \rightarrow B \]
\[ B \rightarrow C \]

We will assume that the reactions are conducted under isothermal conditions, and that the mixture is sufficiently dilute that volume changes on reaction are negligible. A mass balance under these assumptions yields the equations:

\[
\frac{dC_A}{dt} = -r_1 \tag{1}
\]
\[
\frac{dC_B}{dt} = r_1 - r_2 \tag{2}
\]
\[
\frac{dC_C}{dt} = r_2 \tag{3}
\]

where the reaction rates \( r_1 \) and \( r_2 \) are given by the equations:

\[ r_1 = k_1 C_A \tag{4} \]
\[ r_2 = k_2 C_B \tag{5} \]

This complete system of equations forms a simple mathematical model for our reaction system. The goal of this exercise is translate this mathematical model into a format that ABACUSS can recognize, and then use ABACUSS to predict the variation of the concentrations with time as the reaction progresses.

The engineer describes a mathematical model of a chemical process to ABACUSS by preparing an ASCII file in the ABACUSS input language. This is a high level computer programming language specifically designed to express mathematical models of chemical processes. The input language is basically used to describe the equations and variables in a mathematical model to ABACUSS. This capability to add new models as equations makes ABACUSS a very flexible and powerful modeling system. Simulators like ABACUSS are known as equation-oriented or simultaneous process simulators.

ABACUSS input files can be loaded for editing and can also be loaded for translation. It should be noted that these processes are distinct; a file that has been loaded for editing has not been translated (unless this action has been performed as well). To open a file for editing select Open for edit from the File menu. To translate an ABACUSS input file without opening for edit select Translate a file from the File menu, or hit the Translate button above the text edit window. To translate an already opened file hit the Update button. If the translation is successful the text:

Translation successful

will appear in the dialog box at the bottom of the GUI. A tree of the translated files will be available for browsing in the righthand side pane of the GUI (see Figure 2).
The ABACUSS input file for our mass action kinetics example is shown in Figures 3 and 4. It has been copied automatically from the 10.490 course locker to your input directory. It is called Series_Reactions.ABACUSS.

Select File > Open for edit and load the Series_Reactions.ABACUSS into the text editor. Note that the filename is case sensitive because it is a Unix filename. Let’s go through what the input means (see also Figures 3 and 4).

ABACUSS is a case insensitive language: it does not recognize the difference between upper case and lower case characters. In the example input file, the convention has been adopted that keywords of the ABACUSS language are typed in upper case and identifiers introduced by the user are typed in mixed case. Identifiers introduced by the user may be any sequence of the letter characters A–Z, the number characters 0–9, and the underscore character _. The sequence must always start with a letter character, and can have a maximum of eighty characters. Note that any text following a # symbol on a line of input is treated as comment and ignored by ABACUSS.

Variables in an ABACUSS model represent physical quantities (such as the temperature or pressure of the contents of a vessel). Equations (such as a mass balance) relate these variables to form a model of a physical system. All variables in an ABACUSS model must have a type. The type of a variable states certain properties held in common by all variables of that type. A variable type declaration states the following:

- an identifier for the type (for reference in subsequent input)
- a default initial guess for the value of variables of this type (for use in iterative calculations that require an initial guess)
- a lower bound on the value of variables of this type
- an upper bound on the value of variables of this type
- an optional declaration of the units of measurement for variables of this type (currently only used for display purposes)

Any calculation with an ABACUSS model will ensure that the values of variables of a particular
DECLARE

TYPE
  # identifier  # default  # lower # upper
Molar_Consentration = 1.0   : −1E−12 : 1E5   UNIT = "mol/m3"
Reaction_Rate      = 0.0    : −1E−4  : 1E9   UNIT = "mol/(m3 s)"

END # don't forget to mark the end of the DECLARE block

MODEL Series_Reactions # a MODEL must have an identifier for future reference

PARAMETER
  Rate_Constant1    AS REAL # which denotes the type
  Rate_Constant2    AS REAL

VARIABLE
  Concentration_A, Concentration_B AS Molar_Consentration # variable type
  Concentration_C      AS Molar_Consentration
  Rate_1, Rate_2      AS Reaction_Rate

EQUATION
  $Concentration_A = −1*Rate_1 ;
  $Concentration_B = 1*Rate_1 − 1*Rate_2 ;
  $Concentration_C = 1*Rate_2 ;
  Rate_1 = Rate_Constant1*Concentration_A ;
  Rate_2 = Rate_Constant2*Concentration_B ;

END # don't forget to mark the end of the MODEL block!!

Figure 3: ABACUSS II MODEL and DECLARE Blocks for Mass Actions Kinetics Example
Figure 4: ABACUSS II SIMULATION Block for Mass Action Kinetics Example

ABACUSS input is divided into a series of blocks. A DECLARE block is used to declare variable types. Any number of DECLARE blocks may appear in an input file, but a variable type must be declared before it is used. In our example, we need types for molar concentration quantities and reaction rate quantities. The DECLARE block in figure 3 states that variables of type Molar_Concentration will have a default initial value of 1.0, a lower bound of \(-1 \times 10^{-12}\) (concentrations are always positive quantities), an upper bound of \(1 \times 10^5\), and have units of moles per cubic meter. The better the initial guess, and the tighter the bounds you can provide, the better the numerical methods will perform. Note that specifying the bounds too tightly can lead to disaster because the value of a variable will vary with time over a range during a dynamic simulation, and this range must lie within the bounds specified.

MODEL blocks are used to state the variables and equations that make up the mathematical model of a unit operation in a chemical process. Note that a MODEL will typically be an under determined system of equations: there will be less equations than variables. This leaves a subset of variables that must be specified by the user or determined by other models so that the number of unknown variables remaining equals the number of equations. When the number of equations equals the number of unknown variables, the model can be solved (the model is now fully determined). The variables specified to make the model fully determined in this manner are sometimes called the degrees of freedom of the simulation. A MODEL block for our simple example is shown in figure 3.

A MODEL may have a series of time invariant parameters, introduced in the PARAMETER section of a MODEL. This makes a model more general and enables it to be used again in many
applications. For example, a MODEL of a cylindrical vessel that is parameterized by its cross sectional area may be used to represent any cylindrical vessel provided a value for this area is specified. In figure 3 we want to introduce two parameters to represent the rate constants of the two reactions of type REAL. Parameters may also be of type INTEGER and type LOGICAL (i.e., values of TRUE or FALSE). Note that any attribute of a model (e.g., a parameter or a variable) must be declared before it is referred to.

Next, we must introduce the variables that represent the various time dependent quantities in the model. In this case, we want to introduce a variable of type molar concentration for each of the chemical species involved in the reaction: A, B, and C; and a variable for the rate of each reaction. If a set of variables all have the same type, they can just be listed before the type. This is done in the VARIABLE section of figure 3.

The heart of an ABACUSS MODEL are the equations that relate the variables. The EQUATION section of figure 3 shows the equations (1)–(5) transformed into a form that ABACUSS can understand. An equation is two real expressions linked by the equality operator = and terminated by the ; character. Equations may involve constants (e.g., 3.142), parameters, and variables as operands, and the operators +, −, ×, ÷ and ^ (exponentiation – i.e., raising to a power). There are also the built-in transcendental functions shown in table 1. In figure 3 note that the symbol $ is used to denote the time differential operator:

$$\frac{dC_A}{dt}$$

Hence the left hand sides of the first three equations in figure 3 are the accumulation terms in the mass balances.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSOLUTE</td>
<td>The absolute value (magnitude) of the argument</td>
</tr>
<tr>
<td>SIGN</td>
<td>The sign of the argument</td>
</tr>
<tr>
<td>SQRT</td>
<td>The square root of the argument</td>
</tr>
<tr>
<td>ERROR</td>
<td>The error function of the argument</td>
</tr>
<tr>
<td>SIN</td>
<td>The sine of an argument in radians</td>
</tr>
<tr>
<td>COS</td>
<td>The cosine of an argument in radians</td>
</tr>
<tr>
<td>TAN</td>
<td>The tangent of an argument in radians</td>
</tr>
<tr>
<td>ASIN</td>
<td>The arcsine in radians of the argument</td>
</tr>
<tr>
<td>ACOS</td>
<td>The arccosine in radians of the argument</td>
</tr>
<tr>
<td>ATAN</td>
<td>The arctangent in radians of the argument</td>
</tr>
<tr>
<td>SINH</td>
<td>The hyperbolic sine of the argument</td>
</tr>
<tr>
<td>COSH</td>
<td>The hyperbolic cosines of the argument</td>
</tr>
<tr>
<td>TANH</td>
<td>The hyperbolic tangent of the argument</td>
</tr>
<tr>
<td>EXP</td>
<td>The exponential of the argument</td>
</tr>
<tr>
<td>LOG</td>
<td>The natural logarithm of the argument</td>
</tr>
<tr>
<td>LOG10</td>
<td>The logarithm to base 10 of the argument</td>
</tr>
<tr>
<td>INT</td>
<td>Truncate real argument towards negative infinity</td>
</tr>
</tbody>
</table>

Table 1: Table of Built-in Vector Functions

Our model is therefore a system of ordinary differential equations (ODEs) coupled with algebraic equations (AEs). These are usually known as differential-algebraic equations (DAEs).
It is important to note the difference between the equality operator \( = \) and the assignment operator \( := \) in the ABACUSS language. The equality operator is used to denote a general mathematical relationship between two expressions (i.e., an equation), whereas the assignment operator is used to denote the assignment of the value of an expression to a variable; the variable assigned a value appearing on the left hand side of the assignment operator. Examples of the use of the assignment operator appear in Figure 4.

A SIMULATION block is used to specify a particular simulation with a MODEL previously declared. Obviously, many different simulations may be performed with a single model, each simulation being a different scenario in which the physical system represented by the model is studied. An ABACUSS dynamic simulation calls for the numerical solution of an initial value problem (IVP) in the DAEs making up the model. In order to fully specify a simulation, we must make the model fully determined by specifying the degrees of freedom, and define an initial condition for the IVP.

Figure 4 shows the SIMULATION block for our example. First, we must state which MODEL(s) we are going to use for the simulation. The UNIT section creates active instances of the models listed during execution of the simulation.

All the time invariant parameters must be assigned values before the simulation is well-posed. This is shown in the SET section of Figure 4. Note that the particular variables of a model instance are referenced by what is called a pathname mechanism. In its most simple form this is:

Model_Instance_Identifier.Variable_Identifier

In other words, the model instance identifier, a . character, and then the variable identifier. In more sophisticated models (e.g., in the ICE reactor example) there may be a whole list of model instance identifiers separated by . characters prefixing the variable identifier. This occurs when models are nested inside each other to manage the complexity of developing a large process model.

In the first line of the SET section in Figure 4, the parameter is referred to by its complete pathname. In the third through fifth lines, a WITHIN structure is used to define Reactor as the scope within which any pathname is interpreted. This shorthand avoids the need to keep prefixing variable identifiers with model instance identifiers. WITHIN structures may be nested. If an identifier is not found within the current scope, the enclosing scopes are searched sequentially. If the identifier is still not found, the compiler will issue an error message.

The initial conditions of a simulation are defined in an INITIAL section, as shown in Figure 4. The example input states that the initial concentration of species A is 2 \([\text{mol/m}^3]\), and the concentration of the other species is zero (i.e., they are not present initially). In general, initial conditions may be expressed as completely general equations that are added to the MODEL equations in order to calculate consistent initial values for all the variables in the model. Note that the WITHIN structure may be used with equations as well as assignments.

Finally, we need to state how long the simulation should run for. This is done with a SCHEDULE section. In general, the SCHEDULE section may be used to declare very complex sequences of operations to be performed during the simulation (this is discussed below). This feature makes ABACUSS unique amongst process simulators.

An ABACUSS input file may contain any number of DECLARE, MODEL and SIMULATION blocks as required in any order, provided that a block is declared before its identifier is referenced in another block. Your input may also be in several different files, but again, a file containing the relevant declaration must be loaded and compiled before files referring to the block in question can be loaded successfully.

Now that we can understand what the input file means, let’s execute a simulation. Hit the Update button, which will translate the file Series_Reactions.ABACUSS. Loading a file for translation...
will make ABACUSS translate the input file and perform certain checks on the correctness of the input. If there are any syntax errors, etc. in the input file, ABACUSS will inform you that the translation failed in the dialog box at the bottom of the GUI and a window will pop up with the translation errors (see Figure 5). Left-clicking with the mouse on an error on this pop-up window will highlight the line of the input file where the error was found. All errors must be corrected before ABACUSS will allow the input file to be executed. Note, that if a file is already translated, (you can check on the righthand side of the GUI) if you translate the same file again without first unloading it, this will produce errors. The file in the text edit window can be saved and translated by selecting the Update button. This button will also unload the file if it is currently translated.

At this point you should have a correct translated input loaded into the ABACUSS environment. The loaded files are displayed on the righthand side of the GUI (see Figure 2). To execute Series_Reactions_Simulation click on the Execute button. If only one simulation is loaded, then the simulation will execute, otherwise you will have to select which simulation you wish to run. ABACUSS will proceed to set up the problem and integrate the differential equations numerically. For this problem, the simulation will be executed virtually instantaneously. During execution, text output concerning the numerical solution and possible input errors is displayed in the dialog box at the bottom of the GUI. ABACUSS will tell you whether the simulation was successful, or terminated prematurely. The simulation should be successful! You are now ready to display the results generated by ABACUSS, as described in the next section of this memo.
Plotting ABACUSS Results

The best way to view the results of a dynamic simulation is to generate graphs of the variation of the variables with time over the simulation. For plotting graphs use the program \texttt{bgraph} (Figure 6), which can be invoked by clicking the \textit{Plot} button in the ABACUSS GUI, or by typing the following at the Athena command prompt

\begin{verbatim}
athena\% bgraph &
\end{verbatim}

![Figure 6: Plotting environment](image)

The program will read all the available simulation results in the ABACUSS output directory (~\texttt{/ABACUSSII/output}) and load them in a tree form. Files can also be opened explicitly by clicking \textit{Open} button on the toolbar and selecting the corresponding .\textit{VARS} file from a simulation. \texttt{bgraph} also supports plotting data from the .\textit{CSV} output files produced by ABACUSS using \texttt{CSVOUTPUT := TRUE} option in a simulation (see later for details).

When the screen appears, there will be expandable tree in a window on the left, and plotting options on the right. If no trees are visible, that means that no output files have been found, or that they can’t be read. The plotting options on the right have two tabs. The \textit{General} tab is used for setting the most commonly used options and the \textit{Lines} tab for further fine tuning the plot. A variable can be plotted by expanding the trees down to the bottom level and left clicking on the variable name. Note that the plot will pop up in a separate window as shown in Figure 7. You may need to refocus the plot window to see the plot if it was minimized earlier. If you need to plot multiple variables, hold down the \texttt{Ctrl} key and press the left mouse button to choose names of variables required. Each time a new variable is drawn, the scale of the drawing will be recalculated. Because of this, the plotting scale of variables depends on what other variables are plotted.

To enlarge (zoom) a section of a plot, move the mouse to a point which will be the new upper left corner of the plot, and click the right mouse button once. Then move the mouse to the new
lower right corner of the plot and click the right mouse button once and the new plot will be drawn. Each time a plot is zoomed, the boundaries are stored in a stack in memory. Press p on the plot window to recover back the earlier zoom level and n to go forward. Press a for autoscaling the plot. Pressing h on the plot window will print out the list of commonly used commands for the plot window, in the Gnuplot error log box in the right bottom of the main window. If the program appears to freeze after pressing a command key many times on the plot window, pressing h on the plot window will unfreeze it.

From the General options tab on the right, various options for the plot including the title, the axes labels, x-y ranges and line width can be customized. If the plot options are changed and the changes are not reflected in the plot window, press Refresh Plot button on the lower right part of the main window. From the Lines tab, the color of the lines on the plot can be chosen.

In order to customize the legends, scroll the left tree view of the files and variables horizontally. Corresponding to each variable name (i.e., the column Nodename), you will see the legends under the column Plotname. Left click the name and retype the desired name.

Once the plot has been customized to suit one’s taste, a printable version of plot can be created. To create a plot that can be printed on a full page, make sure that under the Edit menu, Landscape EPS is checked and then press EPS button on the toolbar. This will bring up a dialog box asking you to name the file. This way, a color postscript file will be produced. If one chooses colors properly, multiple lines can be differentiated even on a black and white printer. One can also choose to produce a monochrome postscript file by checking the both Monochrome EPS and Landscape EPS option from the Edit menu and finally pressing the EPS button. Each time EPS button is pressed, the program will bring up a postscript viewer (gv) to show the preview of the generated postscript file (which is exactly what will be printed on the printer). To print the postscript files so generated, type the following at the Athena command prompt

```bash
athena% lpr -Pathena_printer_name yourfilename.eps
```
If you want an EPS file to be included in a \LaTeX\ document, you should uncheck the Landscape EPS option from the Edit menu and then press the EPS button.

The graphing program only reads the number of variables (from a .VARS or .CSV file) once, when it is started. Simulation results are read on-the-fly as the variables are plotted. If the simulation results file changes (as a result of a rerun of simulation), pressing Refresh Plot button will refresh the plot to reflect the new values read from the file. If the number of variables changes in a simulation, the graphing program must be restarted.

Additional information about the commands can be obtained by choosing Help from the Help menu or by pressing Ctrl-h on the main window.

**Tutorial Example: First Reactor in ICE Process**

If necessary, unload the input file for the mass action kinetics example. ICE_Reactor1_Model.ABACUSS in the include directory contains the MODEL of the first reactor in the ICE process and the associated feed tanks and valves. Note that these MODELS will never change throughout this project. Hence you should not edit this file! However, it should always be included in files with SIMULATION blocks that represent particular operating policies applied to this physical system.

The physical property models are in the file PhysicalProperties.ABACUSS in the include directory and you will always need to include PhysicalProperties.ABACUSS in the models for the reaction and distillation tasks of the ICE process. At your leisure, take a look at the contents of this file, but you don’t really need to understand it to proceed with the tutorial.

To simulate a particular operating policy for the first reactor, you must write a SIMULATION block describing your operating policy. We have prepared one for you already. Load HW21A.ABACUSS. This file contains the initial conditions, values for the inputs to the system (degrees of freedom), and a schedule that describes any changes to these inputs as well as the conditions that define the end of the simulation. This SIMULATION block represents an operating policy that is uncannily similar to that of the first part of the homework problem. We strongly suggest that you copy this file and edit it in order to experiment with different operating policies for the first reactor.

To perform the simulation, we need to execute the SIMULATION block that was declared in the file HW21A.ABACUSS. The simulation takes about 10 seconds on a pentium III, so be patient. ABACUSS will report on how far it has advanced the simulation time. Once the simulation is complete, you can start to answer the homework problem by using the plot environment. Don’t forget to copy this file before editing it for the other parts of the homework. Note that if you edit a file that has already been successfully translated by ABACUSS, you must use apply the Update command to the loaded file before you run your changes.

**Getting Help**

The most current syntax manual for ABACUSS II is posted at:

http://yorics.mit.edu/abacuss2/abacuss2.html

This contains a detailed description of the complete syntax.

Although ABACUSS II has been thoroughly tested, it may still contain some bugs like all complex software. If you think you have found a bug, please report it to your TA immediately. You should email your TA with the input file that causes the bug, and a detailed description of the sequence of button clicks that causes the bug. We will endeavor to fix the bug as soon as possible.
Note: if a simulation fails to complete successfully, it is most likely due to an error in your input, or some form of numerical failure such as a failure to converge, variables hitting their bounds, etc. Such a numerical failure is not a bug. You will need to reformulate the problem to achieve, for example, convergence. Note also that the GUI has some limitations due to GTK, the toolkit for creating graphical user interfaces.

The output files generated by ABACUSS can get quite large, but you should not get problems with your Athena disk quota, since this is quite large (1GB as of August 2004). In case you do exceed your disc quota, you either have to remove some DATA files

   rm ~/ABACUSSII/output/SIMULATIONNAME.DATA

or store your output in a temporary directory on the hard drive of your local machine. This is done as follows (exit ABACUSS first):

   athena% cd /tmp
   athena% mkdir <your user name>
   athena% cd
   athena% cd ABACUSSII
   athena& setenv AOUTPUT /tmp/<your user name>
   athena% abacussII &

AOUTPUT is a Unix environment variable that stores the output directory for ABACUSS II. The /tmp directory is physically located on the computer that you are working at, and it is erased at midnight every night. So don’t expect your output to still be there if you move to another machine or come back the next day. The simulations should run sufficiently fast that it is no hassle to rerun the simulation to generate new output.

Using existing MODELS

There are many ways of using an existing MODEL. One way is to use INHERITS which copies all the contents of the MODEL in the new MODEL (see Figure 8). Another way is to use a UNIT that is a copy of the MODEL (see Figure 9). Note the different pathnames of the variables in the two options.

Useful Features

Getting numerical values of variables without plotting

The easiest way of getting the value of a few variables at a particular point in the simulation is to use a DISPLAY followed by the path to a variable, as shown for the Simple Mass Action Kinetics.

   ...  
   SCHEDULE
       SEQUENCE
       CONTINUE FOR 25.0
       DISPLAY Reactor.Concentration_A
   END
   END
MODEL Plant1 INHERITS Series_Reactions
#INHERITS makes a copy of the previous model
VARIABLE
  Extra_rate AS Reaction_Rate
EQUATION
  Extra_rate = Rate_1 + Rate_2;
END
SIMULATION ex1
UNIT
  Plant AS Plant1
SET
  Plant.Rate_Constant1 := 0.3;
  Plant.Rate_Constant2 := 0.5;
INITIAL
  WITHIN Plant DO
    Concentration_A = 2;
    Concentration_B = 0.0;
    Concentration_C = 0;
  END
SCHEDULE
  SEQUENCE
  CONTINUE FOR 25.0
  DISPLAY Plant.EXTRA_rate
END
END

Figure 8: Using INHERITS

If you want to access or process all the values of all variables at all simulation times you can incorporate them in a spreadsheet, such as Microsoft Excel. In the OPTIONS section of the SIMULATION block turn the option of CSVOUTPUT (Comma Separated Value) to be TRUE. ABACUSS will write a file called SIMULATION-NAME.csv in the directory ~/ABACUSSII/output. For the Simple Mass Action Kinetics:

SIMULATION Series_Reactions_Simulation # identifier for future reference
OPTIONS
  CSVOUTPUT:=TRUE;
...

ABACUSS will write a file called SERIES_REACTIONS_SIMULATION.csv. By the extension Excel will recognize the format of the file and create a spreadsheet where each column corresponds to one variable and each row to a time value.

Printing the ABACUSS input files

The graphical interface of ABACUSS allows you to print the input files with nice syntax highlighting. From the graphical interface go to File → Print Edited. You will have the options of file or printer, as well as Duplex printing and 2 Up. If you want to save paper use both of them.

Since the ABACUSS input files are ASCII files they can be printed by the lpr command. Alternatively a utility called a2ps which formats and syntax highlights the files can be used.

athena% add daepack
athena% cd ~/ABACUSSII/input/
athena% /mit/daepack/arch/i386_linux24/bin/a2ps -o HW21A.ps HW21A.ABACUSS

takes HW21A.ABACUSS and produces a postscript file, while
MODEL Plant2
UNIT
  React AS Series_Reactions #Reac is an object in Plant2
VARIABLE
  Extra_rate AS Reaction_Rate
EQUATION
  Extra_rate = React.Rate_1 + React.Rate_2;
END

SIMULATION ex2 # identifier for future reference
UNIT
  Plant AS Plant2
SET
  Plant.React.Rate_Constant1 := 0.3;
  Plant.React.Rate_Constant2 := 0.5;
INITIAL
  WITHIN Plant.React DO
    Concentration_A = 2;
    Concentration_B = 0.0;
    Concentration_C = 0;
  END
SCHEDULE
  SEQUENCE
  CONTINUE FOR 25.0
  DISPLAY Plant.Extra_rate
END
END

Figure 9: Using UNIT

athena% add daepack
athena% cd ~/ABACUSSII/input/
athena% /mit/daepack/arch/i386_linux24/bin/a2ps -Pprintername HW21A.ABACUSS

directly prints HW21A.ABACUSS on printername.

Common Errors and Problems in Simulations

All examples in this section are based on the Simple Mass Action Kinetics example.

How to avoid errors in the first place

Both inexperienced and experienced users run into problems when using software for complex models and most of the time it is their own fault, rather than a software bug. Some things to keep in mind for your simulations:

1. If possible, gradually write models or change simulations. After each change run your simulation, and check the results.

2. Take your time when writing the equations (first think, then type).

3. Try to be as explicit as possible (e.g. 50 kmol/hr should be written as 50*1000/3600 and not as 1/0.072 or 13.88889). Put lots of comments explaining to yourself and others changes to the input file.
4. When defining types, be careful with the bounds and default values. Having tight bounds and good defaults helps the solver to converge, but if the bounds are too tight the equations will have no solution.

5. When good guesses for the variable values are available, use them in the `PRESET` section.

6. Remember to take breaks and get enough sleep.

**Variable type already used, model identifier previously used**

If you are trying to load two files which have conflicting models or variables identifiers, ABACUSS will complain in the error log (Figure 11). For big models there will be lots of errors. You have to remove the previous file from translation (`File → Remove from Translation`).

**Could not find a value for . . .**

If you forget to set a value for a parameter, ABACUSS will translate the file without complaints, but when you try to execute it, you will get an error message in the output window. You have to set a value for each parameter (even if the parameter is not used). Figure 12 shows an example of a simulation with a missing parameter value.
SIMULATION no_set # forgot to set a parameter
UNIT
  Reactor AS Series_Reactions
SET
  # Reactor.Rate_Constant1 := 0.3 ; # did not set this
  Reactor.Rate_Constant2 := 0.5 ;

INITIAL
  WITHIN Reactor DO
    Concentration_A = 2 ;
    Concentration_B = 0.0 ;
    Concentration_C = 0 ;
  END
SCHEDULE
  CONTINUE FOR 25.0
END

Figure 12: Forgetting to assign a parameter value

Wrong number of initial conditions, There is 1 overdetermined block, There is 1 underdetermined block

In complex models it is very difficult to have a well defined system of equations. Because of a typo or a conceptual error you may have wrong number (Figure 13) or wrong combination (Figure 14) of equations or initial conditions. ABACUSS will complain about that in the output window.

FIGURE 12: FORGETTING TO ASSIGN A PARAMETER VALUE

Wrong number of initial conditions, There is 1 overdetermined block, There is 1 underdetermined block

In complex models it is very difficult to have a well defined system of equations. Because of a typo or a conceptual error you may have wrong number (Figure 13) or wrong combination (Figure 14) of equations or initial conditions. ABACUSS will complain about that in the output window.

SIMULATION Wrong_init_num # wrong number of initial conditions
UNIT
  Reactor AS Series_Reactions
SET
  Reactor.Rate_Constant1 := 0.3 ;
  Reactor.Rate_Constant2 := 0.5 ;
INITIAL
  WITHIN Reactor DO
    Concentration_A = 2 ;
    # Concentration_B = 0.0 ; # did not provide initial condition
    Concentration_C = 0 ;
  END
SCHEDULE
  CONTINUE FOR 25.0
END

Figure 13: Wrong number of initial conditions

Debugging

ABACUSS has an analysis tool that will help you find which equations are underdetermined or overdetermined. Press the Analysis button. A new window will pop up (Figure 15). ABACUSS divides the analysis task in three steps and each time tries to break down the equation system in as many blocks as possible. You can use the analysis tool even without understanding the underlying mathematics. If necessary press the radio buttons (DAE, Initialization, Index check) and then look at the Underdetermined and Overdetermined blocks. Usually the blocks are not too big and it is relatively easy to find out where the mistake is. An overdetermined block is a system of equations
SIMULATION Wrong_init_comb # wrong combination of initial conditions
UNIT
   Reactor AS Series_Reactions
SET
   Reactor.Rate_Constant1 := 0.3 ;
   Reactor.Rate_Constant2 := 0.5 ;
INITIAL
   WITHIN Reactor DO
      Concentration_A = 2 ;
      Concentration_A=0.6; #thought I should calculate the derivative at TIME=0
      Concentration_B = 0.0 ;#did not provide initial condition
      Concentration_C = 0 ;
   END
SCHEDULE
   CONTINUE FOR 25.0
END

Figure 14: Wrong combination of initial conditions

with more equations than variables; remove one or more equations. An underdetermined block is a system of equations with more variables than equations; add more equations of specify more variables as INPUTs.

Consistent initialization failed(-4) or Integration failed

Common reasons

1. Typo or error in EQUATIONS, SET, INPUT. Example 50 kmol/s instead of 50 kg/hr

2. Big changes in the course of the SIMULATION in the Resets and REINITIAL sections. An example is a step-change in pressure, instead of a ramp-change or feeding a large amount of a reagent instantaneously. Note that changes like that are often physically impossible or very dangerous.
3. Too tight bounds in type definition.

4. Bad initial guesses.

Figure 16 shows an example for a simulation where the consistent initialization will fail, because of a typo that makes a variable value out of bounds.

```
SIMULATION ConsIn # the consistent initialization fails
UNIT
  Reactor AS Series_Reactions
SET
  Reactor.Rate_Constant1 := 0.3 ;
  Reactor.Rate_Constant2 := 0.5 ;
INITIAL
  WITHIN Reactor DO
    Concentration_A = 2e6 ; #2e6 instead of 2
    Concentration_B = 0.0 ;
    Concentration_C = 0 ;
  END
SCHEDULE
  CONTINUE FOR 25.0
END # and, don’t forget to end the SIMULATION block
```

Figure 16: Simulation example where the consistent initialization will fail

**Debugging**

1. Take a break and think about the problem and your input file.

2. Turn on debugging options by pressing the *Debug* button (Figure 17). ABACUSS will write information about which variables in which equations could not be solved in the output window (Figure 18).

3. Change solver options or try a different solver (see **OPTIONS** in the online documentation http://yoric.mit.edu/abacuss2/abacuss2.html).

![Debug Options](image)

Figure 17: Debugging options
Problems with ABACUSS or the Plotting Environment

Freezing of ABACUSS
ABACUSS may seem to freeze when running a big simulation on a slow machine. You will have to be patient and wait for the simulation to end.

Syntax highlighting does not work properly
If you copy and paste in the editor of the input file you might notice that the pasted lines are not highlighted correctly. You have three options:

- Ignore the fact and proceed. Although the lines are not highlighted properly, ABACUSS will understand them correctly.
- Save the file, close it and reopen it. The syntax highlighting will be fine.
- Go to each line that is not properly highlighted and make a change. The syntax highlighting will come back.

There are also some keywords (e.g., DISPLAY) that the editor currently does not recognize.

Find does not work properly
The editor has an option to search for a word within the input file. You can call this by going to Edit → Find (Control-F does not work on all machines). Unfortunately this search option is case sensitive.

Segmentation fault (crash) of ABACUSS

Reasons
1. Pressing buttons or key combinations too quickly.
2. Bug in program, or underlying packages.

What to do
1. Don’t press buttons or key combinations quickly.
2. Often save your files if you make many changes without executing (each time you press the Update button the file is saved anyway).
3. If the segmentation fault is reproducible, contact the TAs with the exact sequence of steps and the files. If there is a bug in the program, the developers might be able to identify and fix the problem. Unfortunately they cannot fix everything, because of some limitations of gtk or the operating systems.

Funny printout on terminal

Sometimes ABACUSS, or the plotting program will write some information on the terminal. As long as the program runs fine you can ignore this information. In case of reproducible failure report them to TAs along with the other information.

The plotting environment seems to freeze

There is a quirk in the bgraph plotting environment associated with the zooming feature in the plot window. If a command key is pressed repeatedly on the plot window (e.g., the plot is fully zoomed out, but you still keep pressing p repeatedly to zoom out), the program will appear to freeze. If this happens, press h on the plot window to unfreeze the program.

Plot window does not show up after selecting more variables/changing options

If you minimized the plot window, it stays minimized. It is still updated by what you do in the main window. So just refocus the plot window after you are done selecting the variables and customizing options. If the plot is still not updated, press Refresh Plot button to refresh plot. Generally it is a good idea to keep the plot window next to the main window and not minimized when you are working with bgraph.

Plotting program has too many trees

You have executed too many simulations. Remove some files from your output directory by typing

```
  athena% rm /ABACUSSII/output/SIMNAME.VARS
  athena% rm /ABACUSSII/output/SIMNAME.DATA
```

or delete all the files in the output directory by typing

```
  athena% rm /ABACUSSII/output/*
```

When I open my file half a line is missing

You probably have a line that is very long and due to a bug in the editor it eats part of the code. Use the carriage return (Enter-key) to distribute the line in many lines.

ABACUSS II Language Reference

This section provides more detailed information on the ABACUSS input language so that you can change input files for the homework problem.
SIMULATION Blocks – Modeling Operating Policies

The conditions characterizing a particular dynamic simulation in ABACUSS are given in the specification of a SIMULATION block. The block refers to the mathematical model that will be used, and adds the information necessary to use the model for a simulation. This information includes the following:

- initial conditions for the simulation.
- a specification of the input variables (of degrees of freedom) for the simulation.
- a schedule that describes any changes to the inputs at later times in the simulation. This is how control actions imposed upon the system are modeled.

The description of a SIMULATION block is divided up into the following sections:

- **OPTION**: this section enables you to set a number of parameters related to numerical options, convergence criteria, output, etc. For details see the syntax manual. You should not need to change these options.

- **UNIT**: this section declares the MODEL blocks that will be used during the simulation.

- **SET**: this section assigns values to any model parameters not already assigned values in the MODEL blocks. ABACUSS will not execute a simulation until all the time invariant parameters are assigned values.

- **INPUT**: this section specifies inputs or degrees of freedom for the simulation. The inputs may be assigned constants or functions of time. The degrees of freedom must be satisfied for ABACUSS to execute a simulation.

- **PRESET**: this optional section allows the specification of initial guesses for some or all of the simulation variables that override the defaults associated with a variable’s type.

- **INITIAL**: this section is used to specify the initial condition of the system for the simulation. ABACUSS will not execute a simulation unless the correct number of initial conditions are specified.

- **SCHEDULE**: this section specifies the stopping criteria as well as any changes to the inputs to the system during the course of the simulation.

You will need to modify the INPUT, INITIAL, and SCHEDULE sections in order to run the distillation and reaction tasks through more sophisticated operating policies.

The INPUT Section

The INPUT section sets the values of a subset of the variables in the process model. For instance, the vapor rate, reflux ratio and column pressure of the distillation tasks are specified as inputs. Note that each of the variables that is specified is referred to by the full pathname of models that denotes its location. The values assigned in the INPUT section may be changed as part of the SCHEDULE; later on in the simulation, these can be changed by using a RESET task that will assign a new value to the variable. Note that the assignment operator (:=) is used rather than the equality operator (=) for all specifications in the INPUT section. The values assigned to a variable in the INPUT section may be expressed as a constant, or a function of time. The keyword TIME is used to denote the time since the start of the simulation (at the start, \( \text{TIME} = 0 \))
**SIMULATION** Series_Reactions_Simulation # _identifier for future reference_

**UNIT**
Reactor AS Series_Reactions

**SET**
Reactor.Rate_Constant1 := 0.3 ;
**WITHIN** Reactor **DO**
  Rate_Constant2 := 0.5 ;
**END** # _within structure_

**PRESET**
**WITHIN** Reactor **DO**
  # _identifier_ # _guess_ # _lower_ # _upper_
  Concentration_A := 2.0 : 0.0 : 5.0 ;
  Concentration_B := 0.0 ;
  Concentration_C := 0.0 : : 5.0 ;
**END**

**INITIAL**
**WITHIN** Reactor **DO**
  Concentration_A = 2 ;
  Concentration_B = 0.0 ;
  Concentration_C = 0 ;
**END**

**SCHEDULE**
CONTINUE FOR 25.0

**END** # _and, don’t forget to end the SIMULATION block_

Figure 19: Using the PRESET Section.

The PRESET Section

In many situations it is desirable to override the default initial guesses (defined in the variable type) for specific variables. For example, we may have more specific information on the value taken by a variable. The better the initial guess, the higher the chance that ABACUSS will converge! Initial guesses and bounds for variables may be changed in the PRESET section, as shown in Figure 19.

The INITIAL Section

The INITIAL section defines the initial conditions for the simulation. The mathematical model of the reactor is a set of differential-algebraic equations, and an initial value problem in these equations is not fully defined without a set of initial conditions. In the SIMULATION block given to you the initial number of moles in the still pot and the accumulators are required as initial conditions. These initial conditions fully define the initial state of the column flowsheet; the model equations define all other quantities in terms of this subset. Note that the equality operator (=) is used
rather than the assignment operator (:=), since ABACUSS treats initial conditions as additional equations that are required to fully specify the system of differential-algebraic equations in order to determine consistent initial conditions.

The SCHEDULE Section

The SCHEDULE section allows the simulation to represent sophisticated operating policies. The SCHEDULE describes the sequence of external actions that are imposed on the model, and varying the sequence of external actions imposed upon the reactor allows us to simulate the reactor with different operating policies. Basically, these features of ABACUSS allow you to experiment with and design an optimal operating policy for the reactor without experimenting with the real plant. Clearly, electronic experiments with ABACUSS are safer, cheaper, and produce no waste when compared with pilot plant experiments.

The distillation and reaction simulations you may want to perform will make use of several features of the SCHEDULE language. These features will be briefly described below, and some of their potential uses will be given. A SCHEDULE block is written in a similar fashion to a computer programming language. The individual statements in a SCHEDULE are known as tasks.

SEQUENCE <list_of_tasks> END The SEQUENCE task encloses a list of other tasks, and specifies that the next task in the list will be started only after the preceding task has been completed. It is used to describe sequences of primitive operations.

CONTINUE FOR <real_expression> The CONTINUE FOR task directs the computer to continue the simulation for the length of time specified by the real expression. The real expression is evaluated at the simulation time at which the task is executed.

CONTINUE UNTIL <logical_proposition> The CONTINUE UNTIL task is used to direct the computer to advance the simulation until the state of the system satisfies the condition defined by the logical proposition. Note that if this condition is never satisfied, the simulation will run forever (or until some limit is reached, such as a tank becoming empty, at which point the simulation fails). The atomic propositions of a logical proposition are relational expressions of the following form:

\[
<\text{real_expression}> <\text{relational_operator}> <\text{real_expression}>
\]

where <relational_operator> is one out of the set of operators \{<,>,\leq,\geq,\neq,=\}. <> means not equal. One has to be careful with the use of <> and = because the simulations are done with finite precision arithmetic on a computer. An example of a relational expression is:

\[
\text{CONTINUE UNTIL } \text{Reactor.Temp} \geq 373.15
\]

A logical proposition may involve one or more relational expressions linked by the logical operators AND, OR and NOT. For example:

\[
\text{CONTINUE UNTIL } (\text{Temp} > \text{Bub.Temp}) \text{ OR } (\text{Temp} < \text{Dew.Temp})
\]

The brackets are not strictly necessary, but they show the logic more clearly. A common problem is not to use the full pathname to the variables in a CONTINUE UNTIL task. For example, you must code:

\[
\text{CONTINUE UNTIL } \text{Plant.ProcessSystem.Reactor1.No.Mols}(7) < 1E-6
\]
You must always use the full pathname in **CONTINUE UNTIL** tasks (this is a feature (!?) of ABACUSS). There is also a hybrid form of the **CONTINUE UNTIL** and **CONTINUE FOR** tasks:

\[
\text{CONTINUE FOR} \ <\text{real_expression}> \ <\text{logical_op}> \ \text{UNTIL} \ <\text{logical_proposition}>
\]

where \(<\text{logical_op}>=\text{AND or OR. The OR will continue for a fixed period, or until the logical condition is satisfied, whichever occurs first. The AND form will continue until both criteria are satisfied. The OR form is particularly useful if you are not sure if a certain logical proposition will ever be satisfied and want the simulation to time out as a precaution.}

**RESET** \(<\text{variable_assignments}>\) \text{END**} The **RESET** task is used to reset the value of variables that have been originally specified in the **INPUT** section at some later point in time during the simulation. For example, a **RESET** task may be used to change the flowrate through a valve by manipulating the variable that represents its stem position. The following would change the stem position of Valve 1 after 3600 seconds:

\[
\text{SEQUENCE}
\text{CONTINUE FOR 3600}
\text{RESET}
\text{WITHIN Plant.Valve_1 DO}
\text{Stem_Position := 1 ;}
\text{END # Within}
\text{END # Reset}
\text{END # Sequence}
\]

This has the effect of opening the valve if the original value for this variable was zero (corresponding to a closed valve). The assignments inside a **RESET** task are expressed in an identical fashion to those in the **INPUT** section.

**REINITIAL** \(<\text{variable_list}>\) WITH \(<\text{equation_list}>\) \text{END**} The **REINITIAL** task can be used to define discontinuities in a set of the model variables. For example, a **REINITIAL** task may be used to indicate the instantaneous addition of a particular reactant to the vessel:

\[
\text{CONTINUE FOR 3600}
\text{REINITIAL}
\text{Plant.Reactor1.No_Mols(8)}
\text{WITH}
\text{Plant.Reactor1.No_Mols(8) = OLD(Plant.Reactor1.no_Mols(8)) + 2 ;}
\text{END # Reinitial}
\]

This task states that variable \text{Plant.Reactor1.No_Mols(8)} is discontinuous, and that its new value is calculated by the equation that follows (i.e., the number of moles of species no. 8 jumps by 2 moles on execution of the **REINITIAL** task). Note the use of the built-in \text{OLD} function to refer to the value of a variable immediately before the discontinuity. In general, the discontinuous variables listed must be differential state variables (i.e., their time derivatives appear explicitly in the process model. For example, mole numbers of each species in the reactor). The number of equations must exactly equal the number of discontinuous variables.

**MODEL Blocks – Modeling Physical Behavior**

This information is here for reference purposes in order to help you understand the input.
Arrays

All the attributes of a MODEL block may be declared as a regular structure, or array, of a base type. Attribute arrays may have an arbitrary number of dimensions. The total number of scalar quantities, or elements, represented by an attribute array is determined from the product of the number of elements in each dimension of that array. The number of elements in each dimension is declared in terms of a scalar integer expression involving integer constants and/or any previously declared integer parameters of the MODEL block in question (e.g. Flow_In AS ARRAY(3,NoStream+1) OF REAL).

References to array attributes may be made in several different fashions. For example, a reference to an entire array is made through use of the attribute identifier alone, and a reference to an individual element of an array is made by an explicit index to the element in question. This index is determined from a list of scalar integer expressions enclosed by brackets following the attribute identifier (e.g. Flow_In(2,NoStream-1)). Each expression in this list represents an index into one dimension of the array. Individual elements of a dimension are indexed from one to the number of elements in that dimension.

A reference to a subset of the elements in one or more dimensions of an array is termed a reference to a slice of that array. The elements that are included within a slice is again determined by a list of references into each dimension of the array in question enclosed by brackets. A subset of the elements in a particular dimension is denoted by two scalar integer expressions separated by a colon, representing the lower and upper bounds of the reference into that dimension respectively (e.g. Flow_In(2:3,1:NoStream)). The value of the upper bound must be greater than or equal to that of the lower bound, and both values must lie with the lower and upper indices of the dimension itself. A reference to an entire dimension is made by leaving a blank, so a list of blanks enclosed in brackets and separated by commas is identical to the use of an attribute identifier alone. A reference to an individual element is again made by a single scalar integer expression (e.g. Flow_In(2:3,1)).

Arrays of equation attributes are not declared explicitly, but are implied by their declaration in terms of expressions involving references to arrays or slices of variable and/or parameter attributes. The dimensionality of a unary expression is the same as that of its operand. For binary expressions, three cases are distinguished:

- if both operands are scalar, then the expression is scalar.
- if only one operand is scalar, then the expression adopts the dimensionality of the other operand. Each element of this expression is obtained by the binary operation between the scalar operand and the corresponding element of the other operand.
- If neither operand is scalar, then both operands must be of the same dimensionality, which is also adopted by the expression itself.

The dimensionality of the equation itself is obtained by applying the rules for binary expressions to the equality operator =.

---

1It is important here to distinguish between the dimensions of an array or regular structure, and the fundamental physical dimensions of a quantity, such as mass or length.

2The dimensionality of any attribute is defined as the number of dimensions and the number of elements in each dimension.

3Each element of this expression is obtained by the binary operation between the scalar operand and the corresponding element of the other operand.

4Each element of the resulting expression is obtained by the binary operation between the corresponding elements of the two operands.
Stream Attributes

*Streams attributes* are subsets, not necessarily disjoint, of the variables describing the time dependent behavior of a system. They represent a system’s interface with its environment, and are useful in specifying the complex connection mechanisms that exist between different components of a physical system.

The **STREAM** section is used to declare stream attributes, which must be declared as instances of already declared *stream types*. This declaration also includes a specification of the subset of variable attributes that is to be included in the stream. The number and types of the variable attributes in a stream must normally match directly those in the stream type declaration. An example of a **STREAM** section is shown in figure 20.

```
STREAM
    Inlet : Flow_In, Temp_In, Press_In, Enth_In AS MainStream
```

Figure 20: Example **STREAM** section

It should be noted that no assumptions concerning the *dimensionality* of the variable attributes included in a stream are made in a stream type declaration. Therefore, a slice or an entire array of variable attributes may appear in any field of a stream attribute, provided the base type of the array matches the variable type of the corresponding field in the stream type. For instance, the following is a valid stream declaration:

```
STREAM
    Inlet : Flow_In(1:NoComp-1),Temp_In,Press_In,Enth_In AS MainStream
```

Stream attributes may themselves be declared as arrays of the basic stream types. For instance, a mixer involving several inlet streams could have a corresponding stream declaration of the form:

```
STREAM
    Inlet : Flow_In, Press_In AS ARRAY (NoStream) OF MainStream
```

Each variable attribute in a \( k \)-dimensional stream must have at least \( k \) dimensions, and each of its *first* \( k \) dimensions must have exactly the same number of elements as the corresponding dimension of the stream. For instance, a possible declaration of the variables in the above example would be:

```
VARIABLE
    Flow_In           AS ARRAY (NoStream,NoComp) OF Flowrate
    Press_In          AS ARRAY (NoStream) OF Pressure
```

This rule allows a natural identification of the variable attributes to be associated with each element of the stream array.

Functions

Expressions may include built-in functions as operands. A function performs a mathematical operation on its arguments that would be difficult or even impossible to declare using the standard language operators. At present, there are two categories of built-in function:
- **Vector functions** take a single argument and return a set of values with dimensionality equal to that of the argument.

- **Scalar functions** take an arbitrary number of arguments of arbitrary dimensionality and return a scalar value.

All function arguments may themselves be expressions of the appropriate type. Table 1 contains a summary of the vector functions currently included in the language definition and table 2 contains a summary of scalar functions.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGMA</td>
<td>The sum of the arguments</td>
</tr>
<tr>
<td>PRODUCT</td>
<td>The product of the arguments</td>
</tr>
<tr>
<td>MIN</td>
<td>The smallest argument</td>
</tr>
<tr>
<td>MAX</td>
<td>The largest argument</td>
</tr>
</tbody>
</table>

Table 2: Table of Built-in Scalar Functions

If any of the arguments of a scalar function are references to an array or a slice, the operation is applied to the entire array or slice. For example, if an array is passed as an argument to the function **SIGMA**, a scalar value equal to the sum of all the elements of that array will be returned (e.g. `Total_Flow_Out = SIGMA(Flow_Out);`). All function identifiers may be used in the declaration of model attributes, thereby locally overriding the built-in function definitions.