ANL-98/8

GCtool for Fuel Cell Systems Design and Analysis: User Documentation

by H. K. Geyer and R. K. Ahluwalia

Technology Development Division

Argonne National Laboratory, Argonne, Illinois 60439
operated by The University of Chicago
for the United States Department of Energy under Contract W-31-109-Eng-38

Reproduced from the best available copy.

Available to DOE and DOE contractors from the
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831
Prices available from (423) 576-8401

Available to the public from the
National Technical Information Service
U.S. Department of Commerce
5285 Port Royal Road
Springfield, VA 22161

ANL-98/8

ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, IL 60439

GCtool FOR FUEL CELL SYSTEMS DESIGN AND ANALYSIS: USER DOCUMENTATION

by

H. K. Geyer and R. K. Ahluwalia

Technology Development Division

March 1998

Work supported by the U.S. Department of Energy under Contract W-31-109-ENG-38.

Table of Contents

GCtool for Fuel Cell Systems Design and Analysis: User Documentation

by

H. K. Geyer and R. K. Ahluwalia

Abstract

GCtool is a comprehensive system design and analysis tool for fuel cell and other power systems. A user can analyze any configuration of component modules and flows under steady-state or dynamic conditions. Component models can be arbitrarily complex in modeling sophistication and new models can be added easily by the user. GCtool also treats arbitrary system constraints over part or all of the system, including the specification of nonlinear objective functions to be minimized subject to nonlinear, equality or inequality constraints. This document describes the essential features of the interpreted language and the window-based GCtool environment. The system components incorporated into GCtool include a gas flow mixer, splitter, heater, compressor, gas turbine, heat exchanger, pump, pipe, diffuser, nozzle, steam drum, feed water heater, combustor, chemical reactor, condenser, fuel cells (proton exchange membrane, solid oxide, phosphoric acid, and molten carbonate), shaft, generator, motor, and methanol steam reformer. Several examples of system analysis at various levels of complexity are presented. Also given are instructions for generating two- and three-dimensional plots of data and the details of interfacing new models to GCtool.

1.0 Introduction

GCtool is a system analysis package similar in scope to that of GPS[1] and GPSTool[2], but instead of using an interpreted PostScript[†]-like language for setting up the system, GCtool makes use of a C-language interpreter. In addition, the interfacing of model libraries to GC has been greatly simplified over that within GPS. As within GPSTool, GCtool has the following capabilities:

- Ability to handle arbitrary system configurations of component models interconnected by various flows.
- Ability to decompose systems into nested or unnested subsystems.
- Ability to impose arbitrary, user-defined system constraints, optimizations, or parameter sweeps over any subsystem.
- Ability to handle components and flows of arbitrary sophistication.
- Ability to permit new components and flows to be added by the user.

GCtool is essentially a C-language interpreter, to which precompiled, user-supplied functions can be linked. The user-supplied functions can represent component models, property codes, mathematical utilities, etc. This permits the computationally intensive aspects of the modeling to be executed at the full speed of optimally compiled coding while still maintaining the flexibility to define problems and execute them using in an interpreter. The interpreted coding is usually small compared with the precompiled code and thus, does not materially slow down the executions. Additionally, the immediate turnaround afforded by the interpreter (i.e., without the need to compile and link as in preprocessor techniques) promotes a rapid prototyping environment for system simulations.

This report discusses the C-like language that can be interpreted (there are a few restrictions as well as a few extensions over standard C), the use of the GCtool environment, the issues pertaining to the interfacing of precompiled C-models to GCtool, the mathematical utilities, and a basic set of component models. The underlying philosophy behind the system analysis is similar to that within GPSTool. Thus, to represent the components within an actual system, one defines instances of the various component model classes. The system configuration is then represented by calling the various functional entries of these component model instances. System tasks, such as executing parameter sweeps, establishing system constraints, or performing optimizations, are set up by defining iterative loops around the system configurations. Each of these concepts will be discussed in more detail in later sections and, in general, is relatively easy to implement. However, as the defining of systems and tasks is done with a C-like language, more effort is required on the part of the user than with a systems code with a fixed input structure. The advantage is that one obtains more flexibility as to the type of problems that can be solved.

The actual type of system problem that can be analyzed with GCtool is dependent on the types of component models that are linked to GCtool. Thus, as with GPSTool, steady-state, dynamic, or even discrete-event models can be used. As part of the GCtool package, a basic set of power system components is supplied along with a mathematical utilities library. Additionally, curve and surface plotting models are furnished.

Section two is a reference section that describes the interpreted language, while section three describes the GCtool environment. The task class that is used to define mathematical tasks within the system simulations is discussed in section four. In particular, section 4.2 shows some simple examples of inputs to GC for solving the basic types of systems tasks upon which more complex simulations are based. Section five discusses the basic set of component models, including the flows that are used, the model functional entries that are used in defining system configurations, and the model parameters. Section six continues the examples started in section 4.2 but includes the use of the component models. Section seven describes the two kinds of plot classes (two and three dimensional), and finally, section eight discusses the model interfacing issues.

[†]PostScript is a registered trademark of Adobe Systems, Inc. PostScript is described in reference [3].

2.0 Interpreted Language

While many system simulations can be set up with only very elementary C coding[†], more complete features are available. The GC language interpreted by GCtool is the same as C in most respects with the changes described in the following.

2.1 GC Types

The basic data types of `int`, `double`, `char`, and `void` are available as are arrays of these types and pointers to these types. At present the `float` type is not implemented. In addition, a `FILE` type has been added without the need to include the `stdio.h` header file. Here are some examples,

```
int i, j, k;
double x, y[25], z[5][10], w;
char str[23];
FILE *fp;
```

The most important restriction concerns the use of pointers. While one can define pointer variables, such as the `fp FILE` pointer above, the GC language has not implemented the indirection operator, *, or the address of operator, &. Thus, the only real use of pointers within GC is where the pointer is used as a variable by itself. For example, the above `fp` pointer is generally only used in opening and closing files, such as with the functions `fopen` or `fclose`, or in the printing or scanning of some file, such as with the functions `printf` or `scanf`, and is seldom used to reference elements of the `FILE` structure itself.

An additional built-in type, denoted as `CFUNC`, is also available and is defined through the following C language:

```
typedef  int (*CFUNC)();
```

Thus, `CFUNC` is used to define a pointer to a function returning an integer. As will be discussed below, GC can directly call precompiled C functions that return an integer. Such functions should be declared within the GC inputs as a `CFUNC` type. The `typedef` statement, itself, is not supported within the GC language.

Variables can be initialized within their declarations by following the variable with an equal sign and the initial value. The initial values may be expressions using any variables that have been previously declared and which have been assigned values. For example,

```
int i=5, j=6*sin(3.0*i);
double x=5.6, y[3]={5.0,4.0,3.0};
char str[10]="a string";
```

Note that for arrays the usual C syntax of placing the array values in `{}` is used. The `braces` are not required when initializing an array of characters. Also, a null character is automatically added if the array bounds are large enough. At present, only one level of `{}` is permitted in array initialization; thus, for multidimensional arrays the initial values will need to be laid out as a one-dimensional array. The order used in this case is the same as with C, with the last array index changing the most rapidly. For example,

```
double y[2][3]={1,2,3, 4,5,6};
char str2[3][4]={"abc","def","ghi"};
```

would initialize `y[0][0]` as `1`, `y[0][1]` as `2`, `y[1][0]` as 3, etc., and `str2[0]` as `"abc"`, `str2[1]` as `"def"`, etc. If fewer initial values are provided than needed to initialize the entire array, then the remaining array values are taken as zeros. Note that the initialization of the character arrays is slightly different, in that the last dimension defines the maximum length of an individual character string. Thus, `str2` above should be thought of as a single three-dimensional array of character strings of a maximum length of four characters, each of which is then initialized by the three string values. Note that when arrays of strings are defined, if an initializer value exceeds the last array bound (i.e., the maximum string length), then a type or size mismatch error message is printed. Note also that if the

[†]The casual user of GCtool can set up many problems without any real knowledge of C by simply following the examples in the later sections. This section, however, will require a knowledge of C to be fully understood.

string needs to be terminated with a null character, space must be provided for it. At present, one cannot initialize an array of characters using only character values. That is, the following

```
char str3[3]={'a','b','c'};   // cannot do
```

is not permitted. Instead, one would have to put all three characters into a string, such as

```
char str3[3]="abc";
```

In this case `str3` is not terminated by a null character since space was not provided for it. Note that due to this character array assignment and the limited use of pointers within GC, it is not possible to write

```
char *str3="abc";     // not allowed
```

to define `str3` to be a pointer to the character string `"abc"`.

Within GC the additional delimiters of `static`, `auto`, or `extern` are not implemented. All variables should simply be treated as if they were `static`. In addition, variables can be only global or local to some function in scope. File scope is not implemented within GC. Global variables are simply those defined outside of any function, and local variables are those defined within a function. As in C, variables (and this includes functions to be discussed below) must be defined before they are used. Within functions local variables must be declared before any executable statement.

In addition to arrays of the basic types, one can also define structures of these types, for example,

```
struct abc
  {double x, y, z[20];
   int i, j;
  };
```

Variables to be defined as these structure types, however, must be defined in a separate statement without the `struct` keyword. Thus, each structure must have a tag name, which is then used like a type to define variables. For example, to define `abc1` and `abc2` to be of type `struct abc`, one would write

```
abc abc1, abc2;
```

Arrays of structures are handled similarly,

```
abc abca[10];
```

Structure definitions cannot be nested within other structures, but, elements within structures may be defined as other structures. In this case the `struct` keyword is optional. For example, one may define a structure `xyz` using the structure `abc` as either

```
struct xyz                  or          struct xyz
  {struct abc a,b;                        {abc a,b;
  };                                      };
```

Structure definitions must also be global in scope, and thus, should not be defined within a function.

The initialization of arrays of structures is not implemented, but one can initialize a single structure using a named initialization as follows. The structure name is followed by an equal sign and then between `{}` braces each element within the structure is initialized by specifying its name, an equal sign, and a value followed by a semicolon.† For example,

```
abc abc1={x=4.5*6.5/3.2;   z={4,3,2,1};   i=4;};
```

Note that only those elements that need to be initialized are specified, that the element names do not have to be in the order of their appearance within the structure, and that the values can be expressions. This named initialization replaces the usual C-language style, where the values of each element in the order that they appear within the structure are placed between the `{}`.

One other aspect concerning structures needs to be mentioned, but is only used when structures are employed with precompiled models. GC will lay a structure out in memory with each of its elements starting on either a byte, word, or double word boundary (depending on the element type) and in the order of the appearance of these elements within

†For backward compatibility with older inputs to GC, comma separators between each initialized item are also acceptable.

the structure. This layout may not exactly match that of a structure that has been generated by a C compiler. Thus, to ensure correct alignment of the structure's elements with that of a precompiled C structure, each element of a structure can be followed (after any array references) by an `@` sign and a byte offset value from the start of the structure. In addition, the total size of the structure in bytes can follow the structure tag name preceded by an `@` sign. This information is usually not supplied by the user but is generated automatically with the interface generator (see section eight).

GC also supports a simple form of the enum keyword, as follows.

```
enum {a, b, c, d, e};
```

Here `a` is defined as the integer `0,` `b` to be `1`, etc.

2.2 Expressions

Expressions within GC are written as in C but with the following precedent orders:

1 . (structure element reference), `[],` `()` (function reference)

2 `*,   /,   ==,   !=,   <=,   <,   >=,   >,   %`

3 `+,   -,   =,   ++,   --,   &&,   ||,   +=,   -=,   *=,   /=`

The grouping of operators is always from left to right; thus,

```
a*b*c
```

is equivalent to

```
((a*b)*c)
```

The precedent order is slightly different from standard C. If in doubt about the precedent order, simply use parenthesis.

The following built-in functions are available and work exactly as their C language counterparts:

```
exp(),      pow(),      sin(),      cos(),      tan(),
log(),      log10(),    asin(),     acos(),     atan(),
atan2(),    fabs(),     ceil(),     floor(),    fopen(),
fclose(),   printf(),   scanf(),    sprintf(),  fprintf(),
fscanf()
```

In the case of the `scanf` and `fscanf` functions, the `&` operator (since it is not available) is not required before the variables that are being assigned values. At present, the format string within the `printf` functions can only use the `'d'`, `'s'`, `'c'`, `'f'`, and `'e'` conversion characters and within the `scanf` functions only `'e'`, `'f'`, `'s'`, `'c'`, and `']'`. These can be preceded by field width, precision, and the other optional modifiers as in C. In particular, when scanning `double`'s, the `l` (long) modifier must be used.

At present the ++ and -- operations can only be used as postfix operators. As a convenience the assignment operator, =, can also be use to assign array or structure elements similar to the way a variable is initialized. For example, if `x` is declared as `double x[5]`, one could write the assignment statement as

```
x={0, 1, 2, 3, 4};
```

Similarly, if `str` is declared as `char str[4]`, one could write

```
str="abc";
```

It is not possible, however, to use the pointer plus offset means of referencing array elements. Thus, one cannot reference the second element of `x` above as *(x+1) since the indirection operator, *, is not available.

Assignments of a whole array or structure can also be made, provided that the type and size of the arrays and structures are the same. Thus, using the structures `abc1` and `abc2` defined previously, one could write,

```
abc2=abc1;
```

Similarly, if `y` is declared as `double y[5]`, similar to the declaration of `x` above, one could write

```
y=x;
```

The logical operators == and != can also be applied to character strings. This extension, along with the above array assignment, permits one to dispense with the usually `strcpy` and `strcmp` functions. Note that the string comparisons need null-character terminated strings to work properly.

2.3 Logical Statements

The standard C language `if` and `else` statements are available; the `switch` statement is not available. For example,

```
if ((x==3) && (y==4))
    {z=sin(4*x); w=6.0*y;}
else
    z=cos(4*exp(-y));
```

Thus, either a block of statements surrounded by `{}` or a single statement (which can be another `if`) can be conditionally executed.

2.4 Looping Statements

The C language `for`, `while`, and `do` statements are available. Thus, for example,

```
for (i=0; i<20; i++)
    {....}

while (x<=sin(2.45))
    {...}


do
    {...}
while (i<=6*j);
```

where in each case the `{....}` represents an arbitrary block of statements. The `{...}` can also be replaced with a single statement. The comma operator is not available; thus, for example, one cannot write

```
for (i=0,j=0; i<20; i++,j++)   // not allowed
```

The `continue` and `break` statements are available to either continue within the same loop or to break out of the current loop as in C.

An additional looping operator, `forall`, has been added which is similar to the `for` operator, but used when the loop counter needs to be incremented by unequal values. The `forall` operator is used as follows:

```
forall (var; val1, val2, val3, val4, val5, val6)
   {...}
```

Here `var` will be defined in turn as `val1, val2,` etc.

2.5 Functions

Functions are defined using a return type, a name, and a parenthesis-enclosed argument list, followed by a procedure block. GC always requires a return type to be explicitly specified as there is no default return type. The `void` type can be used if no returned type is required. Each of the arguments within the argument list must also be preceded by a type. Since the indirection operator is not available to provide for changing the values of passed arguments within the calling function, the concept of a reference type argument, similar to that in C++, has been provided. Thus, argument names preceded by & will represent the actual parameter within the calling procedure. The `return` statement is also available and can be used both with or without some returned expression.

The following code fragments are some examples of functions:

```
void func1(double a, double b)
    {....
```

```
        }

    double func2()
        {double x;
          .....
          return x;
        }

    int func3(int &i, double x, double &y)
        {
         ....
         i=i+1;  y=6.0*i;
         return i;
        }
```

In the last example, `func3`, not only is there a returned value, but the first and last arguments are taken as reference arguments to the corresponding parameters within the call to this function. Thus, any changes to their values in `func3` will be reflected within the calling program. When a reference argument is used, it must correspond to some actual variable in the calling program rather than some expression and match with the type of that variable. Thus, `func3` should not be called as

```
    j=func3( 3*i,  6.0,  sin(3.4) );
```

however, it could be called as

```
    j=func3( i,   exp(-3*sin(2)),   w);
```

where `i` and `w` are known within the calling program and must be an `int` and `double`, respectively.

In addition to functions that are coded in GC and interpreted, functions can also be called that have been precompiled and then linked to the GC interpreter. Such functions should be declared as pointers to functions and are declared within the GC coding without any arguments using the built-in `CFUNC` type as

```
    CFUNC func;
```

When calling these precomplied functions, the pointer to the function is simply called as if it were the function. Thus, one would call `func` above as `func(...)` where ... stands for any arguments and not `(*func)(...)`, as could be done in C.

To limit the very large number of possible interfaces with different return types and different input argument types, the actual coding of the precompiled functions can return, at most, an integer and can take either no arguments, a single `char*` argument, or an array of `char*` arguments. The actual parameters that make up the arguments within the interpreted GC coding are automatically collected and put into this argument array. Thus, for example, if the above C function is called in GC as

```
    func(a,b,i,j,c,k);
```

where `a` and `b` are some structures, `c` is a `double`, and `i`, `j`, and `k` are integers, then the GC interpreter would form an array, declared as `char *args[]`. This array would be filled with the addresses (typed cast to a `char*`) of `a`, `b`, `i`, `j`, `c`, `k`, and 0, and then the function called as `func(args)`. Note that a 0 pointer is passed as the last element of the array for use with functions that are meant to take variable argument lists. When an argument is some expression rather than an actual variable, GC will generate a dummy argument whose value is that of the expression's. At present, the maximum number of arguments that GC will pass is 50. Within the precompiled C function, the user would be responsible for extracting the arguments from the array and appropriately type casting them. Note that since the address of the parameters is passed, the developer of the function can then simulate either pass by value or pass by location. In the case of a function with no arguments, no array argument is passed.

A special case occurs with functions that are declared as members of a structure, that is, using `CFUNC` within the structure. The first argument of such a function will always be a pointer to the structure of which the function is a member. When the function takes only this one argument, this pointer (typed cast to a `char*`) is the only argument passed to the C function. When the function takes multiple arguments, the pointer to the structure will be the first element of the argument array followed, in turn, by the other arguments as additional elements of the passed argument array. On the calling side, this pointer to the structure is never explicitly coded as an argument. Thus, after defining a particular instance, say `abc1`, of struct `abc` in which `fun` is a member function, one can call `fun` as `abc1.fun`. The first argument to this function will be the pointer to the data structure `abc1`. When this is the only argument that the member function requires, then no parentheses and arguments are required when calling the function; thus, `abc1.fun`, not `abc1.fun(abc1)`, is all that is required. Note that this feature of calling a function without argument parentheses is only available for structure member functions, all other functions must have parentheses when called, as in C. If `fun` requires other arguments, say `x`, `y`, and `z`, then it would be called as `abc1.fun(x,y,z)` without structure `abc1` as the first argument.

The actual precompiled C function that is called must be linked in some way to the name that is used for the function within the GC inputs. This is provided by a special function, denoted as `clinker`, which must be provided to the GC interpreter by the user. This will be discussed in the section on model interfacing.

The GC interpreter also performs additional operations when using precompiled C structures. These consist of automatically calling `init` and `term` member functions whenever these are defined within the structure. The `init` function is called whenever an instance of the structure is declared and takes two arguments (as the two elements of the argument array) consisting of the pointer to the structure and the name of the structure specified as a character string. On return, `init` must return the size in bytes of the structure. This size is then checked with the interpreted structure size, and if different, the GC interpreter terminates with a message indicating that the interpreted and compiled structures are not compatible. The C structures used by the precompiled C functions are placed within a header file, which is then simply included within the GC inputs. This header file must reflect the correct structures as used within the C functions. As a convenience to the user, a special interface generator code, GCintf, can be used to quickly scan the C coding and produce this header file. The GCintf code will be discussed in the section on model interfacing. Within the `init` function, any of the elements of the structure can be given default values. The other function, `term`, is called whenever the model structure is deleted. Within the GCtool environment, this occurs right before a new input problem is interpreted. The `term` function can be used to perform any model cleanup functions, such as freeing up space allocated by the model. The `term` function requires only the pointer to the model structure as an input argument. Note that both the `init` and `term` functions are automatically called by the GC interpreter and, thus, generally are never called directly by the user. Essentially, these functions provide for some of the functionality of constructors and destructors used in C++.

2.6 Directive Statements

As in C, a number of statements such as `#include` are also available within GC. These also need to be interpreted, as no preprocessor is used with GC. These statements need not begin in column one as with the C preprocessor and consist of the following:

```
#include "file"
#debug  i
#interrupt
#return
#resume
#delete
```

Note that the # should not be followed by any "white space". The `#include` is used exactly like the C preprocessor directive and simply includes the file, designated by "file", into the inputs at that point. Included files may be nested, but, at present, there should be no more than 10 open files at any one time.

The `#debug` is followed by the number 0, 1, or 2, and is used to turn on (1,2) or turn off (0) various levels of debugging.

The `#interrupt` will cause the GC interpreter to interrupt the current execution at the point where `#interrupt` appears. At that point, the user can input to the console any arbitrary GC coding. An interrupt can also be effected by typing a control-c. If this is done, the current statement being executed within GC is finished before the interrupt takes place. An interrupt of an interrupt results in the program termination.

The `#return` is used to return to the program that called the GC interpreter. Note that in the GCtool environment, discussed in the next section, `#return` simply returns one to the GCtool program.

The `#resume` is used to resume execution of an interrupted program at the point where it was interrupted. Note that any changes to variables within that program made while in the interrupt mode will still be in force.

The `#delete` is used to delete all variables and coding that have been interpreted. In the GCtool environment `#delete` is automatically executed before each new problem is run, so it is seldom ever used directly in the inputs.

2.7 Comments

As in C, comments within GC are delimited by `/*` and `*/`. There is, however, one change over the standard C comments, and that is, if the initial comment delimiter is `/*/`, then the comment delimiter is ignored. That is, the comment is interpreted as if it were GC coding. The reason for this change is that, as will be described in a later section, it is useful to have GC (or more specifically, GCintf) be able to read conventional C code to generate a model header file for use in the GC inputs. Thus, one can use `/*/` and `*/` to delimit comments in a standard C file but still be interpreted as coding by the GC interpreter. The single-line C++ comment style starting with a `//` and ending with a new-line symbol is also supported. Comments cannot be nested.

2.8 Statement Order

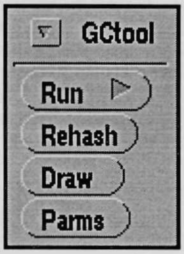
As in C, variables and functions must be declared before being used. If they are not, a message is printed, giving the line number and file where the unknown variable occurred. Also, as in C the local variables within functions must be declared before any executable statements within those functions. Unlike C however, one does not need a main function or even any functions at all. Thus, besides the declaration of global variables, one can also include executable statements outside of any function. This means that statements are simply executed as they are encountered by the GC interpreter. These may be declarations of global variables or functions or executable statements. The GC interpreter does not attempt to locate a main function to start the execution.

3.0 GCtool Environment

GCtool provides a convenient way of running the GC interpreter for performing system simulations. Additionally, GCtool provides a means to quickly query model parameter values, to develop system configuration diagrams, and to assist the user in setting up the GC inputs. At present, GCtool is designed to run on a SUN workstation using OpenWindows Version 3.0. In particular, the GUI makes use of XView. To start GCtool one simply types "gctool" in a cmdtool window within the directory containing the executable at the UNIX prompt. There are three windows within the GCtool environment: main, diagram, and parms.

3.1 Main Window

The main windowcontains the GCtool title and a column of buttons. This window, shown below, is used to invoke the other features of GCtool.



The first button, denoted as `Run`, is actually a menu button with two items, `run` and `run sel`. These menu items are displayed by pressing the right mouse button over the `Run` button. Inputs to GCtool are actually formed in any conventional text editor window and are then selected (i.e., highlighted within the text editor window†). Either of the two `Run` button menu items will then run the selected inputs. The first menu item, `run`, reinitializes the GC environment, that is, deletes any old models previously run, and clears any previously allocated variables, before running the inputs. The second menu item, `run sel`, simply executes the selected text within the context of the last run. That is, the selected text is run without clearing the previous problem. By pressing the left mouse button over the `Run` button, the `run` menu item will be executed without the menu appearing. The second button, `Rehash`, will partially execute the currently selected inputs to determine which models have been used. The third button, `Draw`, will pop open a new window and draw the configuration diagram associated with the currently selected inputs. This window is described in the next section. Finally, the fourth button, `Parms`, will pop open another window, which displays all of the model classes and model parameters. This window also has some features that can be used to help set up new system problems. This window will be discussed in section 3.3.

When a problem is run, the outputs will appear within the cmdtool window in which GCtool has been started. As a means of separating outputs for different runs, if the selected inputs begin with single-line comments (i.e., comments starting with //), each starting in column 0, then these comments will be automatically copied to the outputs at the start of the run.

The documentation that you are currently reading is also available on-line and can be displayed using the standard OpenWindows pageview command by simply placing the mouse cursor within the GCtool window and pressing the h (for help) key from the keyboard. The standard features of pageview can then be used to navigate arround the document.

†Note that GCtool will acquire the selection to run the inputs. Thus, the text editor must have the ability to set a selection and pass it to another application. The standard OpenWindows' Text Editor will do this.

3.2 Diagram Window

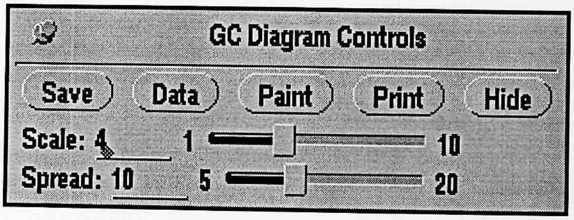
The diagram window that appears when the Draw button is pressed will parse the currently selected inputs and produce a diagram of the system. However, before the diagram will reflect the current inputs, a rehash (or run) needs to be performed so that the models that are being used are known to the interpreter. The diagram is fully editable by the user. The editing that can be done consists of

- Repositioning component models either individually, as a group, or all together,
- Resizing component models either as a group or all together,
- Displaying or hiding component models,
- Changing the layout of the interconnecting flows between models, and
- Displaying state-point information of the flows.

Repositioning, resizing, and hiding all work on groups of models. A model can be assigned to a group by simply clicking the left mouse button on the model. When a model is in a group, a small "+" sign appears in the upper right-hand corner of the model's icon. A model can be removed from the group by the same procedure. All models can be removed from the group by clicking the left mouse button anywhere on the diagram's background, that is, not over any models or flows. To reposition a group of models, press the middle mouse button over any model within the group, drag it to a new location, and release the button. All models within the group will be translated in the same direction and distance that the mouse has been dragged. A single model, not within the model group, can also be repositioned in exactly the same way by using the middle mouse button. Additionally, all models within the diagram can be moved by pressing the middle mouse button on the diagram's background (i.e., not over any model or flow), dragging the mouse to a new location, and releasing the button.

Besides toggling models into or out of a model group, the left mouse button can be used to add or remove kinks within the flow paths. These kinks are indicated by a small circle along the path and only show up on the console and do not appear when the diagram is printed. The kinks can be placed within model groups and repositioned just like any other model. To add a kink, press the left mouse button over the model (or previously generated kink), slide the mouse to the next model (or previously generated kink) along the direction of the flow path, and release the button. The new kink will appear midway along the flow path. To remove a kink, press the left mouse button over the kink, slide the mouse along the flow path to the previous model or kink in the path, and release the button.

To resize models one must open the pop-up control panel. This is done by clicking the right mouse button over the diagram's background away from any model. The control panel, shown below, has five buttons and two slider



items. The first slider, denoted as scale, can be move to resize all model icons. If no model group exists, this action will resize all the models within the diagram. If a model group exists, then only those models within the group will be resized. In the case of a model group, the slider action is slightly different, either increasing or decreasing the model size by roughly 10% depending on whether the slider values is greater or less than five. Thus, repeatedly clicking the left mouse pointer over the slider will either enlarge or reduce the model sizes.

The second slider within the control panel is denoted as spread and is used to spread apart or contract all models within the diagram. This can be useful when adding state-point information to the flows where more space might be needed between the models. Note that the model icons themselves are not resized by this action.

Models can be optionally hidden from the diagram by placing then within a model group and clicking the hide button. This option was provided to hide the less important models in a very complex system diagram. Models that are hidden still show the arrowheads of the incoming flows. Thus, a hidden model can still be placed within a group and then re-displayed by clicking the hide button.

The other buttons within the control panel are used to save the current configuration (`save`), display the state-point information (`data`), repaint the diagram (`paint`, which also turns off the data display), and print the diagram (`print`).

The save button makes use of the file name that appears within the diagram window's title. This file name along with several other variables used by the GCtool is defined by generating an instance of the `modstack` model class within the problem inputs. This instance must be named `mods` for GCtool to locate the file name. Note that this is one of only several times that a model instance must have a specific name, the others being when specific stack classes are used by the models and will be described later. The configuration file name is given by a variable within the `mods` model, denoted as `conffile`. The data button also makes use of a special variable within `mods`, denoted as `rdatfile`, defining the name of the file in which the run data or state point information is placed. The contents of this file can be generated by calling the `mods.rdat` function within the problem inputs. Another `mods` variable, denoted as `caption`, defines a caption that will appear within the configuration diagram whenever the `data` button is pressed. Note that `conffile`, `rdatfile`, and `caption` are all declared as character arrays of length 128, which should be sufficient to hold the corresponding file names or caption. The `mods` structure also holds an array giving the names of the system flow types, denoted as `sysflows`, and another array denoted as `showflows`. The `showflows` array elements are simply 1's or 0's, indicating whether the corresponding flow type within the `sysflows` array should or should not be displayed within the system diagram. More information about the `mods` model will be presented in a later section.

One final feature of the system diagram window is provided. By clicking the right mouse button over any model within the diagram, the parms window, to be discribed in the next section, will pop open (if not already open) and display all of the model's parameters and functional entries. A second click over the model with the right mouse button will hide the model parameters (as described below) but will not close the parm window.

3.3 Parms Window

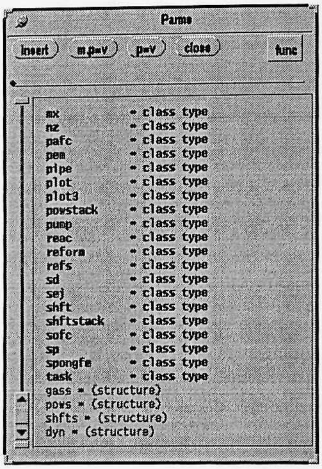
The third GCtool window will appear when the `Parms` button is pressed or, as described above, when a model is clicked on with the right mouse button within the system diagram window. This window will contain a scrolling list of all model classes and model instances that are created for the current systems problem. As with the `Draw` button, the currently selected inputs will need to be rehashed before the scrolling list will display the model instances.

Besides a scrolling list, the parms window, shown below , has five buttons, labeled `insert`, `m.p=v`, `p=v`, `close`, and `func`. In addition, an editable text line appears below the buttons. These items are included so that the user does not have to type model and parameter names when constructing inputs to GC. Thus, they are only a convenience feature and will be described below. The scrolling list initially will display all of the model classes that are available with this version of GCtool. These model classes are listed in bold type and are denoted as the model's class name followed by the designation "= class type". After a rehash or run has been made of any inputs, the scrolling list will also show, after the model classes, all of the model instances that are defined within the inputs. The model instances are shown in normal type followed by the designation " = (structure)". When any of the models is selected, the list will expand to show all of the model's parameters plus their current values. Any parameter that is, itself, a structure or an array will initially display its value by the designation "=(structure)" or "=(array)". By selection of these parameters, the list will again expand, showing the structure's or array's elements. When expanded, the original line previously showing the structure or array designations will change to "= following items". When such a line is selected, the scrolling list will contract back to just the structure or array designation.

Although not associated with the parms window, an additional feature is available for obtaining the values of variables. By selecting any individual variable within the GC inputs and then moving the mouse cursor to the main GCtool window and pressing the space bar on the keyboard, the user can obtain a printout of the variable's value in the cmdtool window in which GCtool was started.

In forming the inputs to GC, the parms window buttons and text item can be used as follows. If a class type item is selected within the scrolling list, a new model instance will be displayed on the text line showing the class type and a default model name. This model name can be edited, if desired, and then when `insert` is pressed, the text line will change to reflect a GC input line necessary to define the model. This text line will also be highlighted so that a simple copy-paste operation† can be performed to insert the line within the input text window. Note that the cursor location within the input text window should be set by the user before the paste operation. When the `insert` button is pressed, the new model instance is also added to the bottom of the scrolling list of the parms window.

If a model parameter is selected within the scrolling list, it will also appear on the text item line. Pressing the `insert` button will strip off the value part, leaving only the fully qualified parameter name (i.e., model name "." parameter name) highlighted. Again a simple copy-paste operation can be used to insert the parameter within the input text window. Alternatively, the `m.p=v` button can be used to insert the fully qualified parameter name along with its value into the text window, or the `p=v` button can be used to insert only the parameter name (excluding the model name) along with its value into the text window. The later two options also include a terminating semicolon since such a line would generally not be in the inputs if not followed by a semicolon. The value of the parameter can be edited before any of the parms buttons are pressed. Note that the short form, generated by the `p=v` button, is appropriate for use in model structure initialization, while the longer form, generated by the `m.p=v` button, can be used to assign model parameter values at other places within the inputs.



```
mx          = class type
nz          = class type
pafc        = class type
pem         = class type
pipe        = class type
plot        = class type
plot3       = class type
powstack    = class type
pump        = class type
reac        = class type
reform      = class type
refs        = class type
sd          = class type
sej         = class type
shft        = class type
shftstack   = class type
sofc        = class type
sp          = class type
spongfe     = class type
task        = class type
gass = (structure)
pows = (structure)
shfts = (structure)
dyn = (structure)
```

The `close` button is used to contract all of the expanded items within the scrolling list. Thus, after `close` is pressed, only the model classes and any model instances will appear within the scrolling list. The `func` button is actually a choice button that will toggle between a pressed or unpressed state and is used to control the information that is displayed within the scrolling list when a model is expanded. When `func` is depressed, the parameter list will only show the model's functional entries. These are listed exactly like they would be used within the system inputs. Thus, selecting one of the entries will cause it to be highlighted on the text edit line, and again, a simple copy-paste operation can be used to insert the entry into the inputs. In this case no other buttons need be pressed. When `func` is not depressed, both the model's parameters and functional entries are displayed within the scrolling list. Thus, `func` is simply used to reduce the number of items in the scrolling list when setting up a system configuration.

[†]Note that this is done by putting the mouse pointer over the Parms window and pressing the *Copy* key and then moving the mouse pointer over the text editor window and pressing the *Paste* key.

4.0 Tasks Involved in System Analysis

4.1 Task Model Class

System analysis involves a number of different tasks: performing parametric studies, establishing system constraints, performing nonlinearly constrained optimizations, integrating sets of equations, etc. The "task model" is used, along with several auxiliary functions, to define and control such tasks. In order to carry out the tasks, a number of mathematical utilities are required, and the task model provides a common user interface to these utilities. The current set of mathematical utilities includes a steepest descent/quasi-Newton update technique for solving systems of nonlinear algebraic equations (partially based on the work in reference [4]), a sequential quadratic programming technique for solving nonlinear constrained optimization problems (based on the work in reference[5]), and the Gear's method for integrating systems of stiff and nonstiff ordinary differential equations [6].

Multiple tasks can be set up in a given systems problem. Each task requires a separate instance of the `task` model class. The task model makes use of auxiliary functions to collect into separate stacks the problem data for the particular task being solved. These data include the variables being varied using the `vary` function, the equality constraints using the `cons` function, the inequality constraints using the `icons` function, the objective functions using the `mini` function, and the differential equations using the `diff` function.

When the controlling function of the task, denoted as `c`, is called, it determines the type of problem that has been set up, allocates the appropriate work space, and then calls the appropriate mathematical utility. While the details of the equation solvers, optimizers, and integrators are beyond the scope of this document, the task model parameters should be understood to effectively use the task class within the GC input. These parameters are described below.

The various `task` functions (`vary, cons, icons, mini`, and `diff`) that are used to set up the task types must lie within a loop controlled by the task's `c` function. This controlling function should be called before any of the auxiliary functions and returns the number 1 if the task is not yet satisfied (i.e., equations not yet solved, integration output time not yet reached, etc.) and 0 when the task is finished. One way to use this function within the GC input is to use the function within a `while` statement, as follows:

```
task a;
while (a.c)
  {
    "task body"
  }
```

Here the task instance is denoted as "`a`", and "task body" will define the problem to be solved using `vary, cons, mini`, etc.

The first auxiliary function, `vary`, requires four arguments. The first is the variable being varied, the second is the starting value for this variable, the third and fourth are values for the lower and upper bounds between which the variable will be constrained. The variable's starting value should be between these bounds. For example,

```
vary(x, 500, 300,  800);
```

would vary the parameter x between 300 and 800 with a starting value of 500. The exact way in which the parameter is varied will, of course, depend on the equation solver or optimizer.

The second function, `cons`, is used to define algebraic constraints or equations that need to be solved. This function requires two arguments. The first is only used to label the constraint and should be a reference to some variable that is not used within any other `cons` function call. Typically, one would use one of the variables within a `vary` call. Note, the label need not have a value or even any meaning for the problem. Its only purpose is to provide a label for the constraint. The second argument is the value representing the residual of the equation to be solved. At the solution this residual should become zero (to within a specified accuracy). Typically, one simply inputs the equation (or rather the algebraic expression representing the equation residual) as this second argument. For example, to define a constraint on x that $e^{-x}x^2 = 0.1$, one could write

```
cons(x, exp(-x)*x*x-0.1);
```

The next function, `icons`, is similar to the `cons` function but is used to define inequality constraints. This function should only be used when one is defining an optimization problem, that is, when the `mini` function is also called within the task body. Here, the second argument represents the inequality constraint residual and at the solution will be constrained to be greater than or equal to zero. For example, to define an inequality constraint on x that $x^2-2x<0$, one would write

```
        icons(x, 2*x-x*x);
```

The `mini` function is used to define objection functions for optimization problems. It requires only one argument representing the objective function value for the optimization task. At the solution this value should represent a local minimum of the objective function. For example to minimize the expression $1 - e^{-x}x^2$, one would write

```
        min(1-exp(-x)*x*x);
```

Since only one objective function can be defined for any one task, no delimiter label is required, as is the case for the `cons` or `icons` functions. Note that to maximize some expression simply minimize the negative of that expression.

The last function, `diff`, is used to define ordinary differential equations for the task. If this function is called, then `vary`, `cons`, `icons`, and `mini` should not be called for this task. It requires two arguments. The first argument is the dependent variable for the differential equation being defined. This differential equation is of the form

```
        dx/dt = f(x,t)
```

Thus, the first argument would be the `x` in this equation. The second argument is the value of `f`. As will be discussed below, the independent variable `t` is represented by the task class variable, `time`. Before entering the task loop, `x` should be assigned a starting value for the integration. For example, if `f` is $x^2 - t$ and the task class instance is defined as `a`, then one would write

```
        diff(x, x*x-a.time);
```

to define the differential equation.

A task body can also be used that does not call any of the auxiliary functions. In this case, the task body is iterated until the maximum number of iterations as defined by the task model parameter, `maxit`, is reached. To terminate the task loop at any time one would simply set the iteration counter, denoted by the model parameter `it`, greater than `maxit`. Alternatively, for the special task, denoted as `dyn`, if no auxiliary functions are called, the `time` parameter is simply increased incrementally by the model parameter, `del`.

The task's c function, in addition to controlling the mathematical utilities, also resets flow stacks that might be used within a system simulation. While this will be discussed in more detail after the model and flows are described in the next section, it essentially permits one to define iterations around any collection of models without having to perform certain model flow initiations at the start of an iterative loop. Thus, if an iterative loop is needed around only a part of the system inputs, it is best to use a loop controlled by a task rather than coding a `for`, `do`, or `while` loop. If the iterative loop extends around all the component model calls within a system input, then a simple `for`, `do`, or `while` loop will not cause any problems since no flow stacks are used before the loop.

The complete list of user-definable variables for the task model is given below. For each variable we indicate whether the variable is an input or output, along with its default value (in parentheses).

it -	Integer defining the current iteration counter value (1000). Input. This parameter is set to 0 within the first call to the task's c function and is changed incrementally by the specific mathematical procedures that are called for the task. For the equation solving and optimization tasks, `it` will have some value equal to or greater than 1000 when the task is finished. The final value is an indication of the type of task termination, with normal termination indicated by `it`=1000. Except for the premature termination of a task loop without any of the auxilary functions, `it` is generally not changed by the user.
maxit -	Integer defining the maximum number of iterations that are allowed in solving equations and in performing optimizations (40). Input. `Maxit` should be less than 1000, as iteration counts greater than that have a special meaning to the equation solver and optimizer.

prt - Integer specifying various amounts of output to be printed during the iterations that the task is performing (2). Input. The value 0 will turn off all printing, requiring that any output be generated explicitly by the GC input. Values greater than zero will produce greater and greater amounts of output. The actual output that is generated is dependent on the task being solved and also requires for its interpretation a greater understanding of the mathematical utilities than can be quickly explained here. However, the default value of 2 provides for a reasonable amount of output for most tasks, and as this is the default, this level of output will be explained in some detail here.

For the equation-solving tasks, the following is obtained. For each iteration, the output will consist of the task name (as furnished by the user within the GC input) labeled as (`task:`), the iteration number labeled as (`n=`), and the square root of the sum of the squares of the constraint residuals labeled as (`f=`). Note that this last value should gradually be reduced to zero as the iterations proceed. Following these values is the list of independent-variable values, i.e., the unknowns of the problem, labeled as (`x=`) and the list of constraint-equation residuals labeled as (`c=`). This last list of numbers should also gradually be reduced to zero as the iterations proceed. Following these items is a line of output giving some values of Newton step norms, steepest descent step norms, etc. Only one of these will be important in most cases, and that is the variable labeled as (`mu=`). This variable gives some measure of the ratio of Newton step versus steepest descent step and will generally be a small number (less than about 3) if the equation solver is not having problems. If `mu` becomes larger (greater than 10), then one should reconsider the problem being solved. For example, it might be singular or not even have a solution.

For the optimization tasks, the outputs give the task name (`task:`) and the iteration number (`n=`). The number of equality constraints (`meq=`) and the objective function value (`f=`) are then given. The next line (`x=`) gives the values of the independent variables. The (`c=`) line then gives the values for the constraints, with the equality constraints specified first, followed by the inequality constraints. Note that unlike the equation solver tasks, the number of independent variables and constraints may be different. A line labeled as (`l=`) gives the value of the termination function (a function similar to the gradient of the Lagrangian only with absolute values within its sums). When this value is less than the specified task accuracy, the problem is considered solved. The value of l is only calculated after a quadratic subproblem has been solved and, thus, does not appear on every iteration. Some of the iterations are line searches, which include an output line that gives the number of the line searches (`nf=`) plus several other parameters pertinent to the line search.

For integration tasks, again the task name labeled as (`task:`) is given followed by the current time (`t=`), the integrator state (`state=`), and integration order (`order=`). The next line labeled as (`x=`) gives the dependent-variable values, and the last line gives the dependent-variable derivatives (`dx/dt=`).

acc - Variable indicating the termination accuracy criteria (10^{-3}). Input. For equation solving tasks, whenever the square root of the sum of the squares of the constraint residuals becomes less than `acc`, the iterations are terminated. For optimization tasks the value of the termination function as specified by the output line labeled as (`l=`) must become smaller than `acc`.

del - Variable indicating the amount of perturbation that the independent variables will undergo when the equation solver or optimizer is calculating gradients of the constraints (10^{-7}). Input. The actual perturbation made in each variable is the maximum of `del` or `del` times the distance between the upper and lower bounds for the variable. At times an equation-solving task may be used within another iterative loop within the GC inputs implying that the task will solve a similar problem again and again. In such cases the `del` parameter can be set to a negative value, which informs the equation solver to make use of the Jacobian that was built up while solving the problem previously. For some problems, this can save significant time. However, this pprocedure must be used with caution, since the current equation solver does not attempt to force the Jacobian approximations to converge to the true Jacobian of the problem at the solution. For the integration tasks, `del` gives the starting integration step size. For the Runge-Kutta integrations (see `meth` below) this step size should be adjusted to a reasonable value for the problem.

meth -	Method flag used by the differential equation integrator, indicating that the Adams-Bashford-Moulton method will be used if `meth=0`; Gear's backward differencing method, if `meth=1`; or a simple fixed-step-size, 4th-order Runge-Kutta method, if `meth=2` (1). Input.
state -	Variable indicating the state of the integrator (0). Input. Initially, this variable is 0 indicating that the integration should be started. On output, it is assigned a value from 1 to 7, indicating the type of step that the integrator is performing. This variable should be manually reset to zero at the start of an integration task if one is performing an iterative loop around such a task. `State` values of 1 indicate that the integrator has reached a specified output time. `State` values of 2 indicate that the integrator has reached a time value for which the dependent variables are known to the requested accuracy. These two values of `state` are the only ones for which it is guaranteed that the time values reached will not become smaller. For all other state values, the integrator may be performing iterations, Jacobian evaluations, or other functions for which a later step might actually be done for an earlier time value. This would be the case, for instance, if the integrator could not maintain the requested accuracy for the current integration step and had to reduce it. This is mentioned because it is often desirable to print out some variables while an integration is being performed, and it is only when `state` is 1 or 2 that the printout of such variables would make sense.
time -	Independent variable used within the integrator (0). Input on the first call. On output, `time` will contain the current time reached during the integration. This variable should also be manually reset if the integration task is repeated within some iterative loop. Note that this variable is denoted as `time` since, very often, time is the independent variable for the integration. This, however, does not preclude using the integrator for integrating over other variables. These variables must just be denoted as `time`.
tout -	Variable indicating the output value to which the integrations will continue (1.0). Input. If several output times are required, the integration task should simply be put within an iterative loop over `tout`. Note that this loop does not repeat the integrations from the starting time, so `time` and `state` should not be reset to zero in this case.

4.2 Task Model Examples

In the following sections several examples are presented that make use of the task model class. The examples presented should give a flavor of the type of problems that can be set up and solved. They illustrate how to solve purely mathematical problems, such as solving equations, performing optimizations, and integrating sets of differential equations. These basic techniques will then be used in later sections with actual systems models to form and solve system constraints, optimizations, etc.

4.2.1 Use of Vary and Cons to Solve a Single Equation

The first example sets up a purely mathematical problem of solving a single equation in a single unknown. The equation is

$$x^2 - e^{-x} = 0$$

Problems such as this are solved by varying the value of x iteratively until the equation is satisfied. Thus, there are three aspects to solving the problem. First, some iterative loop must be defined. This loop will be called the task loop; the task, in this case, is to solve the equation. The task loop will control the iterations and terminate when the task is solved. The second aspect is to define the variable needed to carry out this task and to define a starting value and bounds for this variable. The third aspect is to define the equation to be solved. This equation will also be called the constraint for the task. To specify each of these aspects, the task class c function and one or more of the auxiliary functions will be called. To specify the variable, the `vary` function is used. For specifying the constraint equation, the `cons` function is used. For defining task control, a `task` class instance is defined, and the iterative task loop set up using the GC `while` statement.

The complete GC input necessary to solve the problem is as follows:

```
#include "intf.h"
task a;
double x;
while (a.c)
    { vary(x, 1, 0, 2);
      cons(x, x*x-exp(-x));
    }
```

Here the first statement includes the interface header file that defines all of the currently available model structures.† After the header file is included, the task instance is defined as a. A task body procedure is then defined using a `while` loop. Within the loop the `vary` and `cons` functions are used to define the problem. As indicated previously, the `vary` function takes the name of the variable to be varied, in this case x, followed by a starting value, and lower and upper bounds, here taken as 1.0, 0.0, and 2.0, respectively. The `cons` function takes a variable (for labeling the constraint), here specified as x, and the equation residual. Note that any variables used must be declared as with any C coding. Thus, x is declared as a `double`.

To run this example using GCtool, one would simply start GCtool, and then type the above inputs (minus the `#include` line) into a text editor, select the inputs using the mouse, and then click the left mouse button on the GCtool's Run button. The outputs will appear within the window where GCtool was started. The outputs for this problem, as well as others in this section, are shown in Appendix.

4.2.2 Use of Multiple Vary's and Cons's to Solve a System of Equations

The second example extends the first example to a system of algebraic equations to be solved. For illustrations, suppose these equations are

$$(x-1)^2 - y = 0$$

$$y - 2\ln(e^x + 1) = 0$$

$$z^2 - x = 0$$

Here the GC input would again consist of a single equation-solving task but would include two additional `vary` and `cons` operators to define two additional variables to be varied and two additional equation residuals. Thus, the input is as follows:

```
task a;
double x,y,z;
while (a.c)
    {vary(x, 2, -20, 20);
     vary(y, 2, -20, 20);
     vary(z, 2, -20, 20);
     cons(x, pow(x-1,2)-y);
```

†The inclusion of the interface file is only required when running GC. GCtool will automatically include this file without it being specified within the inputs; thus, in later examples we will not show this line.

```
        cons(y, y-2*log(exp(x)+1));
        cons(z, z*z-x);
      }
    printf("\nx=%.2f y=%.2f z=%.2f", x, y, z);
```

As before, one must decide on some reasonable starting values for `x`, `y`, and `z` and on the upper and lower bounds for these variables. At times this can be difficult, and several values may have to be tried in order to ultimately find a solution. This is especially true if the problem at hand has several solutions, and one is seeking a particular one. In that case changing the bounds may be used to force the equation solver to search for a solution within a particular region. In this case, for lack of more information, the starting value for all three unknowns was taken as 2, and the upper and lower bounds taken as 20 and -20, respectively. Additionally, the `printf` statement was used to print out the final values (however, like the previous example, the default printout at each iteration will also appear).

Since the task's `a.acc` parameter defining the termination criteria was not specified, the default value is used, stopping the iterations when the square root of the sum of the squares of the equation residuals is less then 10^{-3}. If additional accuracy is required, `a.acc` should be made smaller. If substantially greater accuracy is required for more difficult problems, the default maximum number of allowed iterations, currently 40, defined by `a.maxit` will probably need to be made larger.

4.2.3 Use of Multiple System Tasks

The third example sets up precisely the same problem as example two, but in this case, splits the problem into two nested equation-solving tasks. This is to show how complex problems might be decomposed into simpler tasks (although this example is easily solved as a single task). In this case, two class `task` objects, `a` and `b`, are defined, one for each of the two equation-solving tasks.

In the example, `z` will be solved for within the inner task denoted as `b`, and `x` and `y` will be solved for within the outer task denoted as `a`. To reduce the number of iterations to solve the problem, `z` is given the initial value 2 before the task loops are entered. In this way `z` can be initialized to its current value each time that the inner `b` task loop is started. This `z` value will generally be better than simply taking `z` with some fixed starting value. The complete input would be as follows:

```
    task a, b;
    double x,y,z=2;
    while (a.c)
        {vary(x, 2, -20, 20);
         vary(y, 2, -20, 20);

         while (b.c)
            {vary(z, z, -20, 20);
             cons(z, z*z-x);
            }

         cons(x, pow(x-1,2)-y);
         cons(y, y-2*log(exp(x)+1));
        }
     printf("\nx=%.2f y=%.2f z=%.2f", x, y, z);
```

As can be seen in the input, the only change compared to the previous example is the nesting of the inner task loop to solve the equation in `z` within the loop used to solve `x` and `y`. Decomposing a problem into nested problems such as this is often an effective means of solving a problem that seems to be intractable using only one task. Note that, if such a nesting is done, it often helps to keep the tolerance within the inner loops tighter than the outer loops. This is to prevent the inner iterations from washing out the effects of small perturbations of the outer loop variables when gradients of the constraints are being calculated.

4.2.4 Use of Icons and Mini to Solve an Optimization Problem

As a fourth example we show how a nonlinear, constrained, optimization problem can be solved. The problem for illustration is as follows,

```
                minimize   (x-1)2 + (y-2)2 + zez
                with        x, y, z in [0,10]
                such that x-y = 0   and    x-z > 0
```

Again a single task class can be used to solve the problem, along with the `cons`, `icons`, and `mini` functions. The complete input is as follows.

```
task a;
double x,y,z;
while (a.c)
    {vary(x, 1,0,10);
     vary(y, 2,0,10);
     vary(z,3,0,10);
     cons(x, x-y);
     icons(y, x-z);
     mini((x-1)*(x-1)+(y-2)*(y-2)+z*exp(z));
    }
printf("\nx=%.2f y=%.2f z=%.2f", x, y, z);
```

Here, the starting values were taken as 1, 2, and 3 for the three variables. Like the `cons` function, the `icons` function takes a variable (used only to label or delimit this constraint from others) and the constraint residual. For inequality constraints, this residual should be written such that it is greater than or equal to zero. Inequality constraints, of course, will not necessarily be zero at the solution, although they might be. For such optimization problems more inequality constraints can be imposed than the dimension of the problem. The `mini` function is used to inform the optimizer what the objective function is to be.

As with the decomposition used in the third example, additional nested tasks defining other equation-solving tasks can be included to define arbitrary problem types. The optimization tasks, however, should not be nested within other optimization tasks.

Although this problem is relatively easy to solve, with the final solution being obtained in ten iterations, this certainly is not always the case, and several points about optimization problems should be mentioned. First, optimization problems are inherently more difficult to solve than pure equation-solving problems; thus, at times one may need to re-run the problem with different starting points and adjustments in some of the parameters used by the optimizer. One cannot just look at the potential solution and "see" that it is the solution. This is because, looking at the residuals to the constraint equations and noting that the equality and inequality constraints are satisfied is only part of what needs to be considered. At the solution the Kuhn-Tucker conditions should hold. These conditions can only be evaluated by knowing the Lagrangian multipliers and gradients of the objective functions and constraints. Secondly, during the iterations it is quite possible that the value of the objective function may need to increase, for example, when one needs to go "uphill" in order to satisfy the constraints. Thirdly, iterative techniques, like the one being used here, generally only find local minimums. To find a global minimum often requires substantially more work and sometimes requires *a priori* estimates of the second derivatives of the objective functions and constraints. These often are not available. Fourthly, the problem posed may not even have a solution. This may occur, for example, when no feasible region exists for all of the constraints taken together.

With these and other potential problems, several termination messages may occur when defining optimization tasks. The main ones are "initial line search gradient positive", "convergence of independent variables", and "more than 10 function calls in line search". Some of these may indicate that the solution was not found, while in other cases, they may signify that the solution was found but not to the level of accuracy requested. In some cases re-running the problem from a different starting point can resolve the difficulty. At other times this may be the best that can be done with the finite differencing used in calculating the gradients. Sometimes a smaller (or even larger) value of `del` might be tried.

Finally, one may have to decompose the problem, for example, by putting the equality constraints within an inner nested task or even resorting to parameter sweeps rather than an optimization. Sometimes parameter sweeps will give greater insight into the problem under consideration and will indicate that some variables might be eliminated from the optimization problem, thus reducing the dimensionality of the problem.

4.2.5 Use of Diff to Solve a System of Differential Equations

As a final example we set up an integration of three differential equations:

```
dx/dt   =  -x
dy/dt   = y/2
dz/dt   = x-y
```

Again a task is define and denoted as `a`. The default printout defined by the `prt` variable for the task is also set to 0 so that no printout will be generated. To generate several intermediate output values, a sweep is made on the variable defining the output times, `a.tout`, using a `for` loop. Nested within this `for` loop is the task loop implemented using the `while` statement as before. Within the task loop the three differential equations are defined using the `diff` function to indicate the variables being integrated and to specify the right-hand side of the differential equations. After the task the `printf` function is used to print out the values of the time and the three variables. The complete input is as follows:

```
task a={prt=0};
double x=1, y=2, z=0;
for (a.tout=0; a.tout<=5.0; a.tout+=1.0)
    {while (a.c)
        {diff(x, -x);
         diff(y, y/2);
         diff(z, x-y);
        }
     printf("\ntime=%.2f x=%.3e y=%.3e z=%.3e", a.time, x, y, z);
    }
```

Note that although the task's `state` variable would generally be reset to 0 for iterations around an integration task, in this case, the iterations do not start a new task but simply continue the old one to a new `tout` value. Thus, `state` does not need to be reset to 0. Unlike the arbitrary nesting of equation-solving tasks, integration tasks cannot be nested within each other. However, these integration tasks can be nested within or outside of equation-solving or optimization tasks.

5.0 Flow and Model Classes

5.1 Introduction

In this section, we discuss the details of the flow and component model classes. Defined instances of these model classes represent the building blocks of the system. Each of the model classes has a data structure containing the various model parameters. For example, the heat exchanger model class, `hx`, has parameters defining the heat load, hot-and cold-side exit temperatures, heat transfer film coefficients, etc., not all of which would be assigned values or even referred to in any particular systems problem. Additionally, each of the model classes has a number of functional entries, sometimes referred to as "member functions". Most of these are used to process the various flows within the system. Thus, the system configuration is defined through the order in which these model functions are called. For example, the heat exchanger model has `c` and `h` functions used to process the flows on the cold and hot side of the exchanger, respectively. Each of the models also has an `init` function which is used to assign default values to the model's parameters and most models have a `print` function, which is used to print out the results from the model's calculations. Some models also have a `term` function for freeing up space that may have been allocated within the model. None of the model functions (`init`, `print`, or `term`) makes use of flows. As discussed in section 2.5, the `init` and `term` functions are automatically called by the GC interpreter and never by the user.

5.2 Flow Classes

Before discussing the model classes within the next section, some understanding of the flow classes required by the models is necessary. First, a flow class is nothing other than a C structure that contains the information which passes between the various models. These flows will usually represent the variables describing real physical fluids, but can also represent most anything the modeler desires. In general, the user will manipulate the flows of a system by calling the model functions. In particular, for each flow class there is a special model that is used to initialize the flow and place it onto a flow stack. This flow stack is unique for each flow class and is the mechanism by which the flows are passed between the models.

Practically all of the models have as part of their data structure one or more instances of the flow classes. These are used to store the values of the flows, usually at the exit of the model, and can be used in forming constraints and/or objective functions within the GC input. The basic component models make use of two flow classes, denoted as `gastype` and `shfttype`. The special initializing model for the `gastype` flow is denoted as `gas` and will be described below. The flow stack used with the `gastype` flows is denoted as `gass`. The special initializing model for the `shfttype` flow is denoted as `shft`, and its corresponding stack as `shfts`. These also will be described below. As the names suggest, the `gastype` flow is used to represent the flow of gases (or any fluid) between the models, and `shfttype` is used to represent shafts that connect the models.

5.2.1 Gastype Flow Class

The `gastype` flow class is used for representing the flow of fluids (not necessarily a gaseous phase) and has the following variables:

id -	Pointer to the flow's identification
t -	Temperature in K
p -	Pressure in atm
h -	Enthalpy in J/kg
s -	Entropy in J/kg-K
r -	Density in kg/m^3
q -	Quality
m -	Mass flow rate in kg/s
v -	Velocity in m/s
atoms -	Array of atom fractions, kg-atoms/kg of flow, of each element within the flow
comp -	Array of species kg-moles per kg of flow

frozen -	Flag indicating chemical equilibrium status: 1 for frozen chemistry, 0 for equilibrium chemistry

The `id` parameter is used to define which thermodynamic property code is used in calculating the flow's properties. There are actually several thermodynamic property codes available within GCtool. This variable should be assigned one of the character string values "GAS", "STM", "LIQ-species", or "THR-species". Here species is one of the species defined in the `liqdata` file for "LIQ" flows or `thrdata` file for the "THR" flows. For example, `id="`THR-H2`"` defines the properties for hydrogen, and `id="`THR-C8H18-2" defines the properties for 2-methylheptane. The `id` of "GAS" is used to signify that the flow stream is a mixture of gases in chemical equilibrium. The `id` of "STM" represents a flow of water/steam; the `id` of "LIQ-species" is a flow of a pure liquid; and lastly, the `id` of "THR-species" is either a liquid, gas, or two-phase flow of the indicated species. In general flows with different `id`'s cannot be mixed together in those models which mix or combine flows; however, any number of flows with different `id`'s can be used in the same system analysis problem. Several of the models will permit mixing of "STM" or "THR" type flows with those of "GAS", provided each flow has similar species.

In the case where the `id` pointer is assigned "GAS", the actual gas is further determined by the contents of the flows's `comp` array. This array contains the number of kg-moles of each species per kilogram of flow. The actual species that can be used is defined within the `prop.h` file and presently consists of C, CO, CO_2, CH_4, C_8H_{18}, CH_3OH, H, H_2, H_2O, O, O_2, OH, N_2, NO, S, SO_2, H_2S, H_2Oc, CH_3OHc. Here the last two species ending with the lower case c (for condensed liquid phase) represent any liquid water or methanol within the flow. For convenience, the species names (in caps) are defined as a sequence of integers so that the user can refer to a particular species by specifying its name. For example, the kg-moles of CO_2 per kg of flow would be referenced within the gastype's `comp` array as `comp[CO2]`.

For use with "GAS" type flows, the frozen parameter within the flow structure can be set to 1 to prevent the equilibrium code from changing the concentrations. The frozen parameter is the only parameter that the user should ever directly set in the flow structure; all others are defined by the individual models processing the flow. When frozen is set to 1, no chemical reactions take place within the flow; however, phase equilibrium between liquid water or liquid methanol and their vapor states is still permitted. If this phase equilibrium is also not desired, then the gas stack's `noform` parameter can be used (see below).

In addition to the variables, the `gastype` class has several auxiliary functions for determining the properties of the flows. These are generally only used within the model classes and, thus, really don't need to be of any concern to a GCtool user. They would be of concern, however, to a model developer and are discussed in the section on interfacing models with GC. Briefly, these functions consist of `prop` for determining the thermodynamic properties of the flow as a function of `p` and `t`, `p` and `h`, or `p` and `s`; `sat` for determining the liquid and vapor saturation enthalpies;and `atom` for determining the flow's `atom` array. Note that the properties calculations represent a large part of the modeling within many of the component models. For example, often a fluid flow will be taken to some new temperature, enthalpy, or entropy value, and then the properties code called to determine its other properties. When this is done, depending on the flow, the new chemical equilibrium composition, molecular weight of the flow, density, quality, etc., are determined in addition to the temperature, enthalpy, and entropy values. Thus, when discussing the modeling employed, one can generally assume that quantities, such as the flow's molecular weight are known, and no indication is given as to how these are determined since such calculations are performed within the property codes.

5.2.2 Gasstack Class

`Gasstack` is the stack class used with the `gastype` flows. A specific instance of this class is required for any of the models using a `gastype` flow. This instance must be denoted as `gass`. Since it must always be defined, it is pre-defined within the interface header file used by the GC interpreter. In this way the user does not need to explicitly define this instance within the inputs. The `gass` stack itself has several variables, and several member functions. The variables are as follows:

prt -	Print flag (0). Input. When set to one, `prt` is used to print out values of the flow each time the properties code is called. Its use is for debugging.
lowtemp -	Lowest temperature permitted during iterations within the gas properties code (250 K). Input.

thrfsat -	Flag (1). Input. When set to one, `thrfsat`will cause the THR properties code to produce a table of the saturation temperatures as a function of the pressure. This table is then used to calculate the saturation temperature whenever it is needed by the THR properties routines. This is only a performance issue to eliminate the iterations needed to calculate the saturation temperature later on. Note, however, these iterations must be done initially to generate the table.
noform[i]-	Array of integer flags specifying whether (0) or not (1) the i'th species will form when the "GAS" property code is used to calculate chemical equilibrium. The default values are all zeros. `Noform` is useful in eliminating from consideration those species that might exist thermodynamically, but, usually don't appear due to very slow reaction rates. Setting `noform[H2Oc]`, `noform[CH3OHc]`, or `noform[C8H18c]` equal to one will prevent these condensed phases from occurring and, thus, eliminate phase equilibrium calculations for gas streams that have been frozen. `Noform` can also take the value -1, which implies that the species will be frozen during chemical equilibrium calculations. This is useful for freezing certain species but permiting others to react during chemical equilibrium calculations, thus providing a more refined control than the use of the `frozen` parameter within a flow stream.

The `gass` stack member functions include `print`, which is used to print out tables of variables related to the flow state, and `printm`, which is used to print out tables of molar flow rates and mole fractions for individual species. `Printm` is only useful when one or more of the flows have the "GAS" flow id. `Gass` also has `thrwk`, which takes as an argument "RK" or "LK" for setting either the Rudlick-Kwong or Lee-Kesler equation of state, whenever the "THR" property code is used. The default equation of state is the Lee-Kesler. Two other functions are also available within the `gass` class, these are called as follows:

```
gass.sat(fl, hl, hv)
gass.hv(fl, lhv, hhv)
```

where `fl` is a `gastype` flow; `gass.sat` calculates the saturation liquid enthalpy, `hl`, and the saturation vapor enthalpy, `hv`, at the `fl's` pressure; and `gass.hv` calculates the lower, `lhv`, and higher, `hhv`, heating values of the flow. `Gass.sat` should only be used for condensable flows, "STM" or "THR", while `gass.hv` should only be used for flows that can be converted to a "GAS" type flow (i.e., the flow's `comp` array is defined). Note that for `gass.hv`, the heating value is calculated at 298.15 K and 1 atm and only for gaseous flows. Thus, if one desires the heating value for a liquid flow, the energy necessary to vaporize the fluid must to be subtracted from the `lhv` and `hhv` values.

5.2.3 Shfttype Flow Class

The `shfttype` flow class is used to represent physical shafts that enter and leave components. Thus, for example, the compressor model, in addition to having a `gastype` flow representing the inlet and exit gas flows, has a `shfttype` flow used to represent the inlet and exit shafts to the component. The `shfttype` class has the following parameters:

rpm -	Speed of rotation of the shaft, rpm
inertia -	The total inertia of all components on the shaft up to this point within the shaft flow
power -	The total power delivered to the shaft up to this point within the shaft flow

Unlike the `gastype` flow class, the `shfttype` class has no member functions. Additionally, the shaft flows are generally used only when one is performing a dynamic analysis and, thus, may not even be used in many system problems. Most of the models that have functions that deal with the shaft flows can also be run without calling these functions. The `shft` model, discussed below, is used to initiate a `shfttype` flow.

5.2.4 Shftstack Class

The `shftstack` class is used to define the shaft stack for holding the various `shfttype` flows. Like the `gasstack` class, a specific instance of this class is required for the basic components. This instance must be called `shfts` and, as such, is pre-defined within the interface header file. Thus, the user does not need to explicitly define this instance within the GC inputs. The `shftstack` class has no parameters but does have a `print` member function. This function will print out a table of all the `shfttype` flows used by a system, giving the model name, the rpm, shaft inertia of all components on the same shaft up to the model's exit, and the power within the shaft up to the model's exit. Note that the power is algebraic, with components along the shaft either adding or subtracting from the shaft's power. Thus, turbines, motors, etc., would usually add to the shaft power, while compressors, pumps, generators, etc., would usually subtract from the shaft power. At the end of a shaft flow if the power is positive, then the rpm (in a dynamic system run) would tend to increase, and if the power is negative, the rpm would tend to decrease.

5.3 Basic Model Classes

The present collection of component model classes provides a basic thermodynamic description of the component's behavior. Some, such as the steam reformer, provide for detailed calculations of the temperature and flow fields through the device. Models with more process detail can be added by the user if required. The basic classes consist of the following:

Basic Component Model Classes

modstack-	model stack
gas-	gas flow initiator
sp-	gas flow splitter
mx-	gas flow mixer
ht-	gas flow heater/cooler
hx-	gas flow heat exchanger
cp-	compressor
gt-	gas turbine
pipe-	fluid flow pipe
pump-	pump
sd-	steam drum
fh-	feed water heater
cond-	water condenser
df-	diffuser
nz-	nozzle
cb-	combustor
pem-	proton exchange membrane fuel cell
sofc-	solid oxide fuel cell
mcfc-	molten carbonate fuel cell
pafc-	phosphoric acid fuel cell
reform-	hydrocarbon fuel reformer
reac-	generic dynamic flow reactor
shft-	shaft flow initiator
gen-	electrical generator
mot-	electrical motor
dht-	dynamic heater/cooler
dhx-	dynamic heat exchanger
refs-	detailed dynamic methanol steam reformer
cntl-	PID controller
pows-	system powers

5.3.1 Modstack Model Class

As indicated in section 3.2, the `modstack` class is used to store several global variables used by GCtool. It is also used in printing the model outputs by successively calling the `print` functions of all the models defined in the inputs. When used to store the global variables used by GCtool, a special instance of this model class, denoted `mods`, must be defined. Unlike the `gass` and `shfts` stacks that are predefined within the interface header file, `mods` must be defined by the user.

The modstack class has the following parameters:

sysflows[*] -	Array of character strings used to store the names of all the system flow types. Input. This array is used only by GCtool when dealing with system diagrams.
showflows[*]-	Array containing either 0's or 1's indicating whether the corresponding system flow type as defined in the `sysflows` array is to be shown on the system diagram. (Default is 1 for the first flow, indicating the flow is to be shown, and 0 for all others.) Input.

conffile[128]-	Character string holding the file name used in storing the system configuration diagram ("tmp/temp.conf"). Input.
rdatfile[128]-	Character string holding the file name used to store state-point information for the system configuration diagram (""). Input. Note that when `rdatfile` is taken as "" (the default) the `rdatfile` name is made equal to the `conffile` name only with the ".rdat" suffix replacing the ".conf" suffix.
caption[128]-	Character string holding a caption used on the system configuration diagram (""). Input. Multi-line captions can be created by using '\n' character within the caption string. Each occurrence of '\n' starts a new line centered below the previous line.
printer[16]-	Character string defining the printer to be used when plotting the system diagram (""). Input. A null character string defines the default printer for the workstation. This string defines the printer name as it would appear within the -P option of the UNIX lpr command.

The functional entries to the modstack class consist of `rdat` and `print`. Neither of these functions requires arguments and should only be called after all the other models have been called. The `rdat` function will print out to the `rdatfile` the state-point information to be placed on the configuration diagram. The `print` function will successively call the `print` function of all models used in the inputs. The order of these model outputs is the same order as the models are defined within the GC inputs.

5.3.2 Gas (gas) Model Class

The `gas` model class is used to initiate a `gastype` flow. Additionally the `gas` model has a number of member functions for saving and restoring flows from the `gass` stack, closing flow paths, and continuing flows with different property functions. The member function used to initiate a `gastype` flow is denoted as `c`. The `c` member function does not require any input flows, but puts one output `gastype` flow onto the `gass` stack. The modeling begins by simply assigning values to the flow variables as follows:

$$\mathrm{id} = \mathrm{id}_{in}$$

$$\mathrm{m} = \mathrm{m}_{in}$$

$$\mathrm{v} = \mathrm{v}_{in}$$

$$\mathrm{p} = \mathrm{p}_{in}$$

$$\mathrm{comp}_i = \mathrm{comp}_{i,in} \qquad i=1 \ldots \mathrm{NS}$$

where `id` is the flow id as discussed above; `m`, `v`, and `p` are the flow's mass flow rate, velocity, and pressure, respectively; comp_i is the flow's i'th species mole fraction or molar flow rate, and `NS` is the total number of species. The subscript `in` represents input values. Note that `NS` is fixed by the property calculations procedures (`NS` is defined within the prop.h file) and is, thus, not directly input.

The `gastype's` `atom` function is then called to determine the contents of the flow's `atom` array. As an option, if the input mass flow rate is specified as zero, the input $\mathrm{comp}_{i,in}$ array is taken as the molar flow rates rather than the species mole fractions. The input mass flow rate is then calculated from

$$\mathrm{m} = \sum \mathrm{comp}_{i,in}\mathrm{mw}_i$$

where mw_i is the molecular weight of the i'th species.

Next the variables t_{in}, q_{in}, and h_{in} are examined to determine the flow's exit state point. This is done as follows. If the input value of temperature t_{in} is specified as zero and the input value of the flow's quality is greater than -100, then the `sat` property function is called to determine the saturation liquid and vapor enthalpies, h_l and h_v. The flow's enthalpy, `h` is then determined from

$$\mathrm{h} = \mathrm{h}_l + q_{in}\ (\mathrm{h}_v - \mathrm{h}_l)$$

where q_{in} is an input value for the flow's quality. Here the input value of q_{in} should be some reasonable value, usually between 0 and 1, but negative values can represent subcooling and values greater than 1, superheating. If the temperature, t_{in} is non-zero and q_{in} is less than -999, then the flow's enthalpy is simply assigned an input value

$$h = h_{in}$$

The non-zero t_{in} value is then used only to give the property code some reasonable initial guess of the temperature of the flow. If the temperature t_{in} is non-zero and q_{in} is greater than -999, then the flow's temperature is assigned the input value

$$t = t_{in}$$

and the `prop` function is then called to determine the flow's enthalpy. In every case, the `prop` function is called one more time with enthalpy as the input, along with the flow's pressure, to determine the flow's density, entropy, molecular weight, etc.

When the flow's temperature is defined by the input value of t_{in} and is less than the critical temperature of water and the flow's `id` is that of "GAS", the flow can be humidified by specifying a value of the relative humidity ω as follows. First the steam property code is called at the flow's temperature to determine the vapor pressure, p_{vap}, of water. Any water that was input using the `comp` array is removed, and the total molar flow rate mol_{tot} of the dry flow is calculated. The moles of water, mol_{h2o} necessary to humidify the flow to ω is calculated from

$$mol_{h2o} = \omega\, mol_{tot}\left(\frac{p_{vap}}{p - p_{vap}}\right)$$

This flow rate of water is then added to the flow. In this case the mass flow rate out of the model will not be the specified input m_{in} value (or the value determined from the input `comp` array), but something slightly larger. The m_{in} value will represent the flow of dry gases provided that no water was added when the gas composition was defined.

An option is also provided to assign the velocity of the exit flow by using an input flow area, $area_{in}$. If $area_{in}$ is non-zero, then the exit velocity is given by

$$v = \frac{m}{\rho\, area_{in}}$$

where ρ is the calculated density at the exit, and m is the mass flow rate.

The gas model also has a dynamic mode to simulate a liquid storage tank with both an inlet and an exit flow. This is done by defining the differential equations for the total mass of liquid in the tank and the total enthalpy of the liquid in the tank as follows.

$$\frac{dM}{dt} = \dot{m}_0 - \dot{m}_1$$

$$\frac{dH}{dt} = \dot{m}_0 h_0 - \dot{m}_1 h_1$$

where M is the total mass of liquid in the tank, H is the total enthalpy in the tank, $\dot{m}$ is the mass flow rate, h is the specific enthalpy, t is the time, and the subscripts 0 and 1 correspond to the inlet and exit, respectively. The inlet flow quantities are obtained by calling the `cycl` entry (see below) to the model. Once the total enthalpy is known, the specific enthalpy at the exit is determined from

$$h_1 = \frac{H}{M}.$$

This is then used in place of h_{in} to define the specific enthalpy of the flow leaving the `c` entry. An initial value is given to M directly, while the initial value of H is defined by the above equation and the user's input specification of the exit flow at the start of the simulation. Note that no volume considerations are made. That is, it is assumed that the tank can handle any amount of net input flow. The simulations will terminate, however, if the total mass becomes less than or equal to zero.

The parameters to the gas model are as follows, where the default values of the parameters are specified in parentheses, and an indication of whether the parameter is an input is also given:

id -	Gas flow id ("STM"). Input.
m -	Flow rate (1.0 kg/s). Input.
v -	Flow velocity (10.0 m/s). Input.
p -	Flow pressure (1.0 atm). Input.
t -	Flow temperature (298.16 K). Input.
h -	Flow enthalpy (0.0 J/kg). Input.
q -	Flow quality (0.0). Input.
area -	Flow area (0.0). Input.
comp[i] -	Mole fraction of the i'th species, if m was input as a non-zero value. (Default is an array of 0.0's.) Input. If m was input as zero, `comp[i]` inputs the molar flow rate of the i'th species.
humid -	Relative humidity of the "GAS" flow (0.0). Input.
pvap -	Vapor pressure of water at the input flow temperature.
dcompmax -	Maximum of the absolute values of the differences between the inlet-species molar flow rates into the `dcomp` entry and outlet-species molar flow rates from the `c` entry. This value is calculated within the `dcomp` entry (see below) and should become zero for loop closure.
cyclall -	Flag that, when set to 1, informs the `dcomp` function to force a fixed point closure iteration, not only on the `comp` array, but also on `t`, `p`, and `h` (0). Input.
power -	`Powertype` structure, containing the variables `heat` and `work`. Only the `heat` parameter is important for this model (see the `cycl` entry below).
mode -	Character string that when set to "dyn" will perform the calculations far a dynamic liquid tank simulation (""). Input.
mass -	Initial total mass of liquid in the tank when using the "dyn" mode.
fl -	Exit `gastype` flow structure from the model. Note that `fl` needs to be further qualified with one of the gastype parameters, such as, `fl.t`, when used within the GC inputs.
flcycl -	Inlet `gastype` flow to the `cycl` function.
d -	`Gastype` flow structure representing the difference between the flow entering the `cycl` function (see below) and that leaving the `c` function.

As noted above, if the temperature is specified as zero, then the model assumes that the flow is to start at the saturation temperature corresponding to the input pressure. In this case the specified flow quality is used to determine the inlet enthalpy, provided it is greater than -100. Thus, quality set to zero refers to the liquid saturation line, and set to one, the vapor saturation line. Additionally, if quality is less than -999, then the code expects the enthalpy value to be input directly.

In developing system configurations, the various model functions that handle flows are called in the order that the flows need to be processed. At times, it might be necessary to stop the processing of a flow, and at a later point within the configuration restart the flow's processing. This is done by saving the flow and removing it from the flow stack (`gass` in the case of `gastype` flows) and later recovering the flow to be placed back onto the stack. The member function used to save a flow is denoted as `sav`, and the the member function used to recover the flow is `rec`. When a gas model instance is defined for use in saving and recovering a flow, the `print` member function is automatically disabled. Also no input model parameters need to be specified. Note that most system configurations can be defined without the use of the `sav` and `rec` functions, but they might be needed in some very special circumstances.

The member function `cont` is provided to continue a `gastype` flow, but with its properties calculated using another property's code. `Cont` takes one input flow from the `gass` stack and puts one `gastype` flow back onto this stack. When `cont` is used, the model instance only needs input data concerning the new properties. Thus, if switching from a "THR" flow to a "GAS" flow type, only the new `id` of "GAS" and the `comp` array are needed. `Cont` will use the input flow's temperature and pressure. Note that when switching between property codes, the enthalpies, entropies, and densities may not exactly match between the inlet and outlet of `cont`. Thus, switching between property codes should be done carefully, or an energy imbalance could arise within the system. The "GAS" properties code assumes ideal gas states for the species and, as such, switching between property codes is best done at high temperatures where the species behavior is closer to the ideal gas.

The member function `cycl` is provided to help set up system constraints when a flow forms a closed cycle. This function requires one input flow from the `gass` stack and calculates the differences in the flow variables between this input flow and the output flow from the corresponding `c` function. The differences are stored in the `gastype` flow structure denoted as `d (e.g., d.t, d.p, d.m, etc.)`. This function will also calculate the difference in power (mass*enthalpy) between these two flows and save this in the variable `power.heat`. Note that for a correctly formulated closed path, this variable should be zero. As discussed above, this function is also used in the "dyn" mode to represent the inlet to a liquid storage tank.

One last member function is provided, denoted as `dcomp`, and can also be used to help form a closed flow path. This function will make changes to the model's `comp` array in order to match flow rates for the input and output species. The input flow used is that to the `cycl` function; thus, the `cycl` function must be called first. The `dcomp` function takes the input flow's `comp` array and calculates the corresponding `comp` array used to define the outlet flow from the model's `c` function. Within the GC inputs, one can simply form a task loop with no auxiliary function calls and iterate around the system with `dcomp` called at the end of the task. In this way, a simple fixed-point iteration scheme is implemented to force compositional closure. Note that this function should not be used within a task loop that is making use of the `vary` or `diff` functions. The `dcompmax` parameter can be examined on each iteration to terminate the loop (i.e., setting the task's `it` parameter greater than the `maxit` parameter) when sufficiently close to convergence. The `cyclall` parameter can be set to one to tell `dcomp` to also do a fixed-point iteration on the mass, temperature, pressure, and enthalpy, otherwise these parameters will need to be closed by some other means. Note that more than one flow can be closed in the task loop, however, since only a simple fixed point scheme is being used here, one may need to experiment to close the flows.

5.3.3 Mixer (mx) Model Class

The mixer (`mx`) model class is used to mix together two `gastype` flows using the member function `c`. This function takes one input flow from the `gass` stack and puts one output flow back onto the stack. The other input flow is obtained by calling the member function `s`. This function, which must be called before the `c` function, takes one input flow from the `gass` stack, but generates no output flows. The model requires no input parameters.

Generally, the mixer model should only be used to mix flows that use the same property `id`'s. Thus, "GAS" flows should not be mixed with flows of other `id` types. However, when a "GAS" flow has a species that is the same as the species within a "THR" or "STM" flow, the `cont` entry of the `gas` model can be used to convert the flow to a "GAS" flow for mixing within the `mx` model. Alternatively, the mixer model will permit mixing of flows of different `id`'s, provided that one flow is a "GAS" `id`, and the other is either "STM" `id` or a "THR-species" `id` in which the species is one of those present within the "GAS" flow. In these cases the non-"GAS" flow must have its `comp` array defined at the point where it enters the mixer. Note that the `comp` array is generally only used with the "GAS" type flows, but is required for the other flows in this special case.

In the case where one of the input flows is a "GAS", and the other is a "STM" or "THR", the flow that is a "STM" or "THR" is essentially converted to a "GAS" before being mixed with the other flow. This is done by summing the inlet enthalpies of the two flows and then calling the property code to establish the temperature and other state variables of the exit flow. Since the input "STM" or "THR" flow might have been in the subcooled or two-phase region, the resulting enthalpy of the mixed flow might be such that the resulting temperature is very low, possibly lower than the `gass` stack's `lowtemp` parameter. In this case, the temperature output from the mixer model will be wrong, even though the output enthalpy is correct, and any further heating of the flow by downstream models will again produce the correct temperature. Note that this mixing of "GAS" and "STM" or "THR" flows relies on the property codes using the same reference values for the enthalpies of the individual species within the flows. This reference state is the enthalpy of formation of the individual species at 298.15 K and 1 atm. Using this reference state for the individual species also permits the mixer to calculate the correct adiabatic flame temperature for various mixed flows.

The modeling within the mixer is dependent on whether the input flows are "GAS" flows. For such flows, the output `comp` array must be first calculated from the two inlet `comp` arrays, as follows:

$$\mathtt{comp}_i = \frac{\mathtt{comp}_{1,i}\mathtt{m}_1 + \mathtt{comp}_{2,i}\mathtt{m}_2}{\mathtt{m}_1 + \mathtt{m}_2} \qquad \mathtt{i=1\ ...\ NS}$$

where comp is the composition, m is the mass flow rate, and the subscripts 1 and 2 correspond to the two input flows. Once the $\mathtt{comp}_i$ values of the output flow are known, the atom function is then called to determine the flow's atom fractions.

For both "GAS" and non-"GAS" type flows, the following calculations are then made to determine the output flow's pressure, enthalpy, and mass flow rate:

$$\mathtt{p = min\ (p_1\ ,\ p_2)}$$

$$\mathtt{h = (m_1\ h_1\ +\ m_2\ h_2)\ /\ (m_1\ +\ m_2)}$$

$$\mathtt{m = m_1\ +\ m_2}$$

Finally, the prop function is called with enthalpy as an input to determine the flow's entropy, density, and temperature.

The mixer model parameters are

fl -	Exit gastype flow from the model. As with all model flows, fl would need to be further qualified, such as fl.t, when used within the GC input.
fls -	Secondary input flow to the model.

5.3.4 Splitter (sp) Model Class

The splitter (sp) model is used to split a gastype flow into two flows using the member function c. This function takes one input flow from the gass stack and places one output flow back onto the stack. The second output flow can be obtained by calling the member function s. This function places one output flow onto the gass stack but requires no input flows. Since the second output flow can only be known after processing the input flow through the c function, the s function should only be called after the c function is called.

The modeling done within the splitter is dependent on whether the splitter is being used to split off certain species (only used for flows with an id of "GAS") or to split the whole flow. If the split ratio value, sr, is less than zero, then it is assumed that at least one element of the species split ratio array, $\mathtt{ssr}_i$, is non-zero. In this case the mass flow rates of each species must be calculated to determine the split-off flow. This is done as follows:

$$\mathtt{m}_{2,i} = \mathtt{ssr}_i\ \mathtt{comp}_{in,i}\mathtt{mw}_i\ \mathtt{m}_{in} \qquad \mathtt{i=0\ ...\ NS}$$

$$\mathtt{m}_{1,i} = (1-\mathtt{ssr}_i)\ \mathtt{comp}_{in,i}\mathtt{mw}_i\ \mathtt{m}_{in} \qquad \mathtt{i=0\ ...\ NS}$$

where $\mathtt{m}_{1,i}$ and $\mathtt{m}_{2,i}$ are the mass flow rates of flow 1 and 2 for individual species, $\mathtt{comp}_{in}$ is the species moles per kg array for inlet flow, $\mathtt{mw}_i$ are the molecular weights for individual species, and $\mathtt{m}_{in}$ is the mass flow rate for the inlet flow. Once the mass flow rates are known for the individual species, the total flow rates for the two flows can be determined:

$$\mathtt{m}_1 = \sum \mathtt{m}_{1,i}$$

$$\mathtt{m}_2 = \sum \mathtt{m}_{2,i}$$

The new species moles per kg arrays for each flow can then be determined, as follows:

$$\mathtt{comp}_{1,i} = \frac{\mathtt{m}_{1,i}}{\mathtt{mw}_i\mathtt{m}_1} \qquad \mathtt{i=0\ ...\ NS}$$

$$\mathtt{comp}_{2,i} = \frac{\mathtt{m}_{2,i}}{\mathtt{mw}_i\mathtt{m}_2} \qquad \mathtt{i=0\ ...\ NS}$$

The atom function is then called to determine the atom fraction array for each flow, followed by a call to the prop function with the inlet temperature as input to determine theenthalpy, entropy, and density for each flow.

Note that when the splitter model separates one or more species from a flow, it is really modeling a possibly complex process that may not be simultaneously isothermal and isoenthalpic, which is what the splitter model is assuming. Thus, when splitting off species, one should be aware that it could give rise to an energy imbalance. When such is the case, it may be more appropriate to model the split-off process by more than just a single splitter.

When `sr` is non-negative, the above calculations are replaced by the following:

$$m_2 = sr\ m_{in}$$

$$m_1 = m_{in} - m_2$$

In this case there is no need to call the `prop` function as the state variables for the exit flow are the same as the inlet flow.

The splitter model parameters are the following:

sr -	Split ratio representing the fraction of input mass flow rate that is split off to form the second output flow (0.5). Input.
ssr[i] -	The i'th species split ratio representing the fraction of the input mass flow rate of the i'th species that is split off to form the second output flow. Input. Actually, `ssr` is an array of values but, like the gastype `comp` array, individual species are specified by name, for example, `ssr[CO2]`.
fl -	Primary flow structure from the model. Output.
fl2 -	Secondary or split-off flow from the model. Output.

The `ssr` array is only used with flows having the "GAS" `id` and only when the `sr` parameter is less than zero. Since both `sr` and `ssr` represent fractions of the input flow mass, their values should be between 0 and 1. The `ssr` array elements should not be all zeros or ones, as this would make one of the output flows have a zero mass. Note that since the `sr` variable is by default greater than zero, if `ssr` is to be used, `sr` must be explicitly set to a negative number.

5.3.5 Heater (ht) Model Class

The heater (`ht`) model class is used to transfer heat into or out of a `gastype` flow. The model has one main calculation function `c`. The function takes one input flow from the `gass` stack and puts one output flow back onto the stack.

The model first calculates the exit flow pressure (`p`) based on an input pressure fraction (f_p), as follows:

$$p = p_{in} - f_p p_{in}$$

where p_{in} is the inlet flow pressure. The model then calculates the enthalpy change based on one of three options. If the exit flow temperature `t` is specified as non-zero, then the exit flow enthalpy is simply calculated using the `prop` function with `p` and `t` as inputs. If `t` is zero, then the model checks the exit flow quality, `q`, and if that variable is greater than -100, then the `sat` function is used to determine the saturation liquid and vapor enthalpies, h_l and h_v, at the exit flow pressure. These values are then used to calculate the exit flow enthalpy from

$$h = h_l + q(h_v - h_l)$$

Finally, if `q` is not greater than -100, the heat transferred, Q, is used to determine the exit flow enthalpy from

$$h = h_{in} + \frac{Q}{m}$$

where h_{in} is the inlet flow enthalpy, and `m` is the mass flow rate through the heater. Once the enthalpy of the flow is known, `prop` is called to determine the temperature, entropy, and density of the exit flow. In addition, the heat transferred from either the input or from

$$Q = \frac{(h - h_{in})}{m}$$

is stored for later print out.

The heater model parameters are defined as follows:

temp -	Temperature of the exiting gas flow (500 K). Input.
qual -	Quality of the exiting gas flow (-1000). Input.
pfrac -	Fraction of the input pressure used as a pressure drop (0.0). Input.
heat -	Heat input (W). Input.
power -	The powertype structure, where power.heat represents the heat transferred across the heater. Output.
fl -	Exit flow from the model. Output.

Only one of temp, qual, or heat should be input. Temp is used if not equal to zero. If temp is zero, then qual is used if greater than -100 . In this case the exit temperature will be the saturation temperature at the exit pressure if qual is between zero and one. Note that qual can be set less than zero to represent subcooled flow, or greater than one for superheated flow. If either temp or qual is used to determine the exit flow temperature, then heat is an output variable. Finally, if temp and qual are not used (i.e., set to 0.0 and -1000, respectively), then heat is used directly to determine the exit temperature. Note that heat can be a negative number, in which case this model will act like a flow cooler.

5.3.6 Compressor (cp) Model Class

The compressor (cp)model class is used to model a multistage gas flow compression process. The model has both a design and off-design mode. In the off-design mode the model makes use of performance maps. The performance maps are obtained by calling the in member function, which will read the maps from a file. This in function will be called automatically in the off-design mode within the main calculational function. Alternatively, the user can explicitly call the in function. The main function is denoted as c and performs the gas compression process. This c function takes one gastype input flow from the gass stack and puts one output flow back onto the stack. An additional entry is provided, denoted as cool, which also takes one gastype flow from the gass stack and puts one output flow back onto the stack. This function is called to represent the cooling flow through intercoolers. The intercoolers are considered as part of the compressor model whenever the number of compressor stages is greater than one. For use in dynamic system studies, a shft function can be called to obtain the input shaft flow to the compressor. If called this function takes one shfttype flow on the shfts stack and should be called before the c function. The c function will then put the exit shfttype flow onto the shfts stack. If the shft function is not called, then the model will make use of the rpm with the model's input parameters rather than the rpm from the shaft flow.

The modeling used in the compressor is dependent on whether the model is being called in the design or off-design mode. If in the design mode, the model simply uses a specified exit pressure (p_{ex}) and input efficiency (η). A stage pressure ratio (pr) is then calculated from

$$pr = \langle p_{ex}/p_{in}\rangle^{\frac{1}{n}}$$

where p_{in} is the inlet pressure, and n is the number of compressor stages. Rated (or design point) parameters are then calculated for use in the off-design mode. These include a pressure ratio, efficiency, corrected mass flow parameter ($cmass_{rated}$) and corrected rpm parameter ($crpm_{rated}$) and are obtained from

$$pr_{rated} = pr$$

$$\eta_{rated} = \eta$$

$$cmass_{rated} = m_{in}\frac{\sqrt{t_{in}}}{p_{in}}$$

$$crmp_{rated} = \frac{rpm_{in}}{\sqrt{t_{in}}}$$

where $\mathtt{m_{in}}$ is the inlet mass flow rate, $\mathtt{t_{in}}$ is the inlet flow temperature, and $\mathtt{rpm_{in}}$ is the rotation speed of the inlet shaft.

If the model is run in the off-design mode, a corrected mass flow parameter ($\mathtt{cmass}$) and corrected rpm parameter ($\mathtt{crpm}$) are calculated from

$$\mathtt{cmass} = \frac{\frac{\mathtt{m_{in}}}{\mathtt{p_{in}}}\sqrt{\mathtt{t_{in}}}}{\mathtt{cmass_{rated}}}$$

$$\mathtt{crpm} = \frac{\mathtt{rpm_{in}}}{\mathtt{crpm_{rated}}}/(\sqrt{\mathtt{t_{in}}})$$

The performance maps are then called with these two corrected parameter values to obtain a stage pressure ratio ($\mathtt{pr_{map}}$) and an efficiency ($\eta_{\mathtt{map}}$). In order to make use of the same performance maps for different-sized compressors, these returned map values are further scaled as follows:

$$\mathtt{pr} = 1 + (\mathtt{pr_{map}} - 1)\frac{(\mathtt{pr_{rated}} - 1)}{(\mathtt{pr_{rated,map}} - 1)}$$

$$\eta = \frac{\eta_{\mathtt{rated}}}{\eta_{\mathtt{rated,map}}}\eta_{\mathtt{map}}$$

where $\mathtt{map}$ refers to the quantity obtained from the map, $\mathtt{rated}$ refers to the input rated quantity, and $\mathtt{rated,map}$ refers to the quantity from the map at the rated conditions.

The rest of the modeling is the same in both the design and off-design modes. For each compressor stage the $\mathtt{pr}$ value is first used to determine the stage exit pressure

$$\mathtt{p_{ex}} = \mathtt{pr} \times \mathtt{p_{in}}$$

Then a call to the $\mathtt{prop}$ function is made with the inlet entropy as input to determine the enthalpy ($\mathtt{h_s}$) of an isentropic compression to this exit pressure. The actual exit flow enthalpy ($\mathtt{h}$) is then determined from

$$\mathtt{h} = \mathtt{h_{in}} + (\mathtt{h_s} - \mathtt{h_{in}})/\eta$$

and another call to $\mathtt{prop}$ with enthalpy as the input will then determine the rest of the state points of the exit flow.

The power required by the compressor stage is calculated as

$$\mathtt{Pow} = \mathtt{m(h_{in} - h)}$$

Between each stage intercooling is calculated by cooling the compressed flow down to an intercooler exit temperature equal to the inlet temperature plus some $\mathtt{dt}$ value. The enthalpy of the compressed flow is first saved as $\mathtt{h_{ex}}$, then the $\mathtt{prop}$ function is called with temperature reassigned as the intercooler exit temperature. The stage intercooling heat load is then calculated as

$$\mathtt{Q} = \mathtt{m(h - h_{ex})}$$

where $\mathtt{h}$ is the enthalpy of the compressed gas but at the intercooler exit temperature. The values of the flow's enthalpy, entropy, pressure, and temperature are then saved as the inlet conditions for the next stage, and the above calculations are repeated for all stages. The total power required is calculated as the sum of all stage powers, and the total intercooling heat load, as the sum of all intercooling loads.

When the model is used in a dynamic system analysis, the above set of calculations is used, only now the input rpm comes from the input shaft flow as obtained from the $\mathtt{shft}$ function. The exit shaft flow inertia and power are then increased by the inertia and power of the compressor (note that, in this case, the power is negative).

There are two performance maps for the compressor, both stored in the same input file. The first supplies a pressure ratio as a function of the corrected mass and corrected speed, and the second gives the efficiency as a function of the corrected mass and corrected speed.

An approximate weight of the intercooler heat exchanger is calculated from

$$W = 0.5AT\rho$$

where W is the weight, A is the heat transfer surface area calculated from

$$A = \frac{Q}{u\Delta t_{mean}}$$

ρ is the material density, T is the thickness of the exchanger walls, u is the heat transfer coefficient, and Δt_{mean} is the log mean temperature difference across the exchanger.

The compressor model parameters are defined as follows:

file -	File containing the off-design performance maps ("cp.map"). Input.
mode -	Design("d") or off-design("o") mode specification("d"). Input.
eff -	Efficiency (0.85). Input in the design mode.
pres -	Exit pressure (5.0). Input in the design mode.
pr -	Pressure ratio across a compressor stage. Output.
nstages -	Number of compressor stages (1). Input.
rpm -	Revolutions per minute (5000). Input.
inertia -	Polar moment of inertia for the compressor (5.0). Input.
tin -	Inlet temperature to the first intercooler stage. Output.
tout -	Exit temperature from the intercooler. Output.
dt -	Approach temperature between an intercooler stage outlet temperature and the inlet gas temperature to the compressor (5 K). Input.
lmtd -	Log mean temperature difference across an intercooler stage (K). Output.
rat_cm -	Rated or design-point value of the corrected mass flow parameter. This value is obtained by running the model in the design mode; in which case, this parameter becomes an output value.
rat_crpm -	Rated value of the corrected rpm parameter. This value is obtained as with the `rat_cm` parameter.
rat_pr -	Rated value of the compressor pressure ratio (outlet to inlet) (5.0). Input.
rat_eff -	Rated value of the compressor efficiency (0.85). Input.
power -	Power (watts) required by the compression process. Note that the power is treated algebraically, with negative values representing power consumed by the model. Thus, in a normal compressive process this parameter will be negative.
cm -	The corrected mass parameter. Output.
crpm -	The corrected rpm parameter. Output.
u -	Overall heat transfer coefficient within the intercoolers (30.0). Input.
area -	Total heat transfer surface area within the intercooler. Output.
power -	`Powertype` structure. Output.
fl -	Exit gas flow from the model.
shftf -	Exit shaft flow from the model.

5.3.7 Gas Turbine (gt) Model Class

The gas turbine (`gt`) model class is used to model a gas-flow expansion process. Like the compressor model, `gt` has both design and off-design modes, and in the off-design is based on performance maps. The performance maps are obtained by calling the `in` member function, which will read the maps from a file. In the off-design mode this `in` function will automatically be called from within the main calculational function, provided the user has not previously called `in`. The main function, calculating the gas expansion, is denoted as `c` and takes one `gastype` input flow from the `gass` stack and puts one output flow back onto the stack. A second function denoted as `s` is also available and is used to obtain any extracted or split-off flow. This entry requires no input flow but puts one output flow onto the `gass` stack and should only be used after the `c` function has been called. For use in dynamic simulations, a `shft` entry may be called. If called, this function requires one `shfttype` flow from the `shfts` and should be called before the `c` function. The `rpm` used within the model is then obtained from the shaft flow rather than through the model parameters. If the `shft` entry is called, the `c` function will also put one `shfttype` flow onto the `shfts` representing the exit shaft flow.

The gas turbine model is very similar to a one-stage compressor model, the only difference being the calculation of the exit flow enthalpy, which for the turbine is given by

$$h = h_{in} - \eta(h_{in} - h_s)$$

and the calculation of the exit pressure

$$p = \frac{p_{in}}{pr}$$

where the notation is the same as that used in the compressor model. Any extracted flow that is required is calculated as follows.

$$m_{ext} = f\ m$$

where `f` is an specified input fraction of mass in the extracted flow. The exit mass flow rate of the main flow, m_{ex}, is then redefined as

$$m_{ex} = (1-f)\ m$$

The other state points for the extracted flow are the same as for the exiting main flow. This extracted flow is saved and is placed on the `gass` stack by calling the `s` function.

As with the compressor model, the gas turbine model has are two performance maps, both stored in the same input file. The first supplies a pressure ratio as a function of the corrected mass and corrected speed, and the second gives the efficiency as a function of the corrected mass and corrected speed.

The gas turbine model parameters are defined as follows:

file -	File containing the off-design performance maps ("gt.map"). Input.
mode -	Design ("d") or off-design ("o") mode specification ("d"). Input.
eff -	Efficiency (0.85). Input in the design mode.
pres -	Exit pressure (1.0). Input in the design mode.
pr -	Pressure ratio across the turbine. Output.
rpm -	Revolutions per minute (5000). Input.
ext -	Fraction of input mass flow rate split off into an extraction flow (0.0). Input.
rat_cm -	Rated or design point value of the corrected mass flow parameter. This value is obtained by running the model in the design mode; in which case, this parameter becomes an output value.
rat_crpm -	Rated value of the corrected rpm parameter. This value is obtained as with the `rat_cm` parameter.
rat_pr -	Rated value of the turbine pressure ratio (inlet to outlet) (5.0). Input.
rat_eff -	Rated value of the turbine efficiency (0.85). Input.
power -	`Power` structure, where `power.work` represents the work generated by the expansion process. Note that the power is treated algebraically, with negative values representing power consumed by the model. Thus, in a normal expansion process this parameter will be positive.
cm -	The corrected mass parameter. Output.
crpm -	The corrected speed parameter. Output.
fl -	Exit gas flow from the model.

5.3.8 Heat Exchanger (hx) Model Class

The heat exchanger (hx) models the transfer of heat from a hot gastype flow to a cold gastype flow. This is done using two member functions, h for the hot side and c for the cold side of the exchanger. Both of these member functions take one input flow from gass stack and put one output flow back onto the stack.

As with the heater model, on entry to either the hot- or cold-side functions, the model first calculates the exit flow pressure (p) based on an input pressure fraction (f_p) as follows:

$$p = p_{in} - f_p p_{in}$$

where p_{in} is the inlet flow pressure. Here f_p may be specified differently on either side of the heat exchanger.

The model has several options and makes use of either t_{cold} or t_{hot} to specify the exit flow temperature (t) on either the cold or hot sides. The particular variable that is specified should refer to the function that is called first within the GC inputs, either h or c. Thus, one has either

$$t = t_{cold}$$

$$Q = m\ (h - h_{in}\ )$$

or

$$t = t_{hot}$$

$$Q = m\ (h_{in} - h)$$

where h_{in} in the inlet flow enthalpy on the appropriate hot or cold side, and the exit flow enthalpy h is determined from a call to the prop function with the temperature as input. Once Q is known, it is used on the other side to calculate the exit flow enthalpy, which, in turn, determines the other exit flow properties using a call to the prop function.

Alternatively, one can make use of either q_{cold} or q_{hot} to specify the exit flow quality on either the cold or hot sides. In this case t_{cold} and t_{hot} should both be set to zero. Again, the particular variable that is specified should refer to the function, either h or c, that is called first within the GC inputs. In this case the exit enthalpy is determined from

$$h = h_{in} + q\ (\ h_v - h_l\ )$$

where h_v and h_l are the vapor and liquid saturation enthalpies at the flow's pressure, respectively, and q is either q_{cold} or q_{hot}. The heat transferred, Q, is then determined as before.

As an additional option, Q can be input directly. In this case, both t_{cold} and t_{hot} should be set to zero, and q_{cold} and q_{hot} should be less than -100.

Once both sides of the heat exchanger model have been called, the log mean temperature difference is calculated from stored values of the inlet and exit temperatures and the following equation:

$$\Delta t_{mean} = \frac{x - y}{\log\left(\frac{x}{y}\right)}$$

where x and y are the inlet and exit fluid temperature differences of the heat exchanger. Note that for the purpose of using Δt_{mean} in system constraints, if either x or y or both become less than zero, a fictitious value of Δt_{mean} is returned, although one that still shows the correct trend as a function of x and y. Based on specified values of the heat transfer coefficients on the hot and cold sides, u_{hot} and u_{cold}, an overall heat transfer coefficient is determined from

$$u = \frac{1}{\left[\frac{1}{u_{hot}} + \frac{1}{u_{cold}}\right]}$$

The heat transfer area is determined by

$$A = \frac{Q}{u\Delta t_{mean}}\ .$$

As a convenience to the user in setting up off-design calculations, the heat transfer area can also be input, with the model calculating a constraint residual as

$$c = Q - uA\Delta t_{mean}$$

The c value would still need to be driven to zero by some outside interative loop.

An approximate heat exchanger weight is calculated using

$$W = 0.5AT\rho$$

where W is the weight, T is the wall thickness, and ρ is the wall material density.

The heat exchanger model parameters are defined as follows:

t_cold -	Exit temperature of the cold side (0.0 K). Input.
t_hot -	Exit temperature of the hot side (0.0 K). Input.
q_cold -	Exit flow quality of the cold side (-1000). Input.
q_hot -	Exit flow quality of the hot side (-1000). Input.
heat -	Amount of heat transferred from the hot to the cold flows (0.0 watts). Input.
pf_cold -	Pressure drop fraction on cold side (0.0). Input.
pf_hot -	Pressure drop fraction on hot side (0.0). Input.
ufh -	Heat transfer film coefficient for the hot side (0 watts/m^{2}K). Input.
ufc -	Heat transfer film coefficient for the cold side (0 watts/m^{2}K). Input.
lmtd -	Log mean temperature difference across the exchanger.
mode -	Either 'd' or 'o', indicating that the model is to calculate the heat transfer area, or that this area is to be input and the constraint residual as defined above is to be calculated ('d'). Input.
area -	Heat transfer surface area (m^2). Output.
cons -	Constraint residual between the heat and that calculated from the input area.
thickwall -	Wall thickness (0.001 m). Input.
denswall -	Wall material density (7800 kg/m^3). Input.
type -	Character string indicating the type of heat exchanger, "count" for counter flow or "paral" for parallel flow ("count"). Input.
flc -	Exit flow on cold side.
flh -	Exit flow on hot side.

Only one of `t_cold`, `t_hot`, or `heat` should be input to the model. If either `t_cold` or `t_hot` is used, then that side of the heat exchanger should be called first. These parameters are used to determine the value of `heat`, which then becomes an output parameter. If both `t_cold` and `t_hot` are zero, then the values of `q_hot` or `q_cold` are used to determine the heat load, provided they are greater than -100; otherwise the value of `heat` is used directly to determine the exit conditions.

5.3.9 Pump (pump) Model Class

The pump model (pump)class represents a simple compression process of a liquid flow to a specified pressure at a specified efficiency. Note that this model assumes that the liquid is almost incompressible (constant density) and, thus, should only be called where the flow is in the liquid region. The model has one calculational member function, denoted `c`. This function takes one input flow from the `gass` stack and puts one output flow back onto the stack.

The modeling consists of the following equations:

$$Pow = m\frac{(p_{in} - p_{exit})}{\rho\eta}$$

$$h = h_{in} - Pow / m$$

$$p = p_{exit}$$

where Pow is the power required, p_{in} is the inlet pressure, p_{exit} is the specified exit pressure, ρ is the fluid density, m is the mass flow rate, p is the exit flow pressure, h_{in} is the inlet enthalpy, h is the exit flow enthalpy, and η is the specified efficiency. Once the exit flow pressure and enthalpy are known, a call to prop with enthalpy as the input determines the exit flow temperature and entropy.

The pump model parameters are defined as follows:

pres -	Exit flow pressure (20.0 atm). Input.
eff -	Efficiency of the compression process (0.85). Input.
power.work -	The work required (watts) to accomplish the pumping action. Output. Like the compressor model, work consumed in the compression process will be indicated by a negative value of this parameter.
fl -	Exit flow from the model. Output.

5.3.10 Pipe (pipe) Model Class

The pipe model class represents fluid flow through a pipe. The model has one member function, c, which takes one input flow from the gass stack and puts one output flow back onto the stack. The model can handle a simple pressure drop or, by specification of a number of nodes, both a pressure drop and a thermal time delay for use in dynamic simulations.

On entry to the c function, the model calculates the pressure drop from a friction factor (f) given by Churchill:

$$f = 2\left(\left(\frac{8}{Re}\right)^{12} + \frac{1}{(a+b)^{1.5}}\right)^{\frac{1}{12}}$$

where

$$a = \left(2.457\log\left(\frac{1}{\left(\frac{7}{Re}\right)^{0.9} + 0.27\varepsilon}\right)\right)^{16}$$

and

$$b = \left(\frac{37530}{Re}\right)^{16}$$

Here Re is the Reynolds number based on the hydraulic diameter of the pipe, and ε is the equivalent sand grain roughness height divided by the hydraulic diameter. In calculation of the Reynolds number, the fluid viscosity is obtained from

$$\mu = \mu_{ref}\left(\frac{T}{T_{ref}}\right)^{N_{ref}}$$

where T is the fluid temperature, and μ_{ref}, T_{ref}, and N_{ref} are specified input values. The pressure drop is obtained from f using

$$\Delta p = \frac{2f\rho v^2 L}{D}$$

where ρ is the fluid density, v is the fluid velocity, L is the pipe length, and D is the hydraulic diameter.

If the pressure drop option is used, indicated by specifying the model's nnode parameter as zero, the exit pressure is then calculated as

$$p_{exit} = p_{inlet} - \Delta p$$

The other fluid properties are then calculated by using this pressure and the inlet enthalpy through a call to prop. If the multi-node option is in effect, nnode>0, then the above pressure drop is divided equally between the nnode nodes. In addition, the following differential equation of the enthalpy balance is solved over each node:

$$\rho V \frac{\partial h_{exit}}{\partial t} = \dot{m}(h_{in} - h_{exit})$$

where h_{exit} and h_{in} are the exit and inlet enthalpy values for each node, $\dot{m}$ is the mass flow rate, and V is the node volume.

The pipe model parameters are defined as follows:

diam -	Pipe diameter (0.1 m). Input.
area -	Pipe flow area (m^2). Output.
vol -	Pipe total volume (m^3). Output.
length -	Pipe length (1 m). Input.
fric -	Friction factor. Output.
re -	Inlet pipe Reynold's number. Output.
mu -	Fluid viscosity. Output.
muref -	Fluid viscosity at the reference temperature ($2.671x10^{-5}$). Input.
tref -	Reference temperature for visocity expression (500 K). Input.
nref -	Exponent in the visocity expression (0.6364). Input.
rough -	Sand grain roughness/hydraulic diameter for the pipe (0.0). Input.
nnode -	Number of nodes used (0). Input. Nnode should be less than or equal to 9.
h[10] -	Array of node exit enthalpies. Output.
t[10] -	Array of node exit temperatures. Output.
fl -	Exit flow from the model.

5.3.11 Diffuser (df) Model Class

The diffuser (df) model class represents a gaseous flow diffuser. The diffuser model has one calculational member function, denoted c. The model takes one input flow from the gass stack and puts one output flow back onto the stack.

The model has both a design and an off-design mode. On entry to the model, the total pressure, p_t, of the flow is determined by iterating on the pressure at constant inlet entropy until a value of the enthalpy equal to the total inlet enthalpy (h_t) is obtained, where

$$h_t = h_{in} + v_{in}^2/2$$

and h_{in} and v_{in} are the inlet enthalpy and velocity, respectively. In the design mode, once this total pressure at the inlet is known, the exit values for the velocity, enthalpy, and pressure are then determined from

$$v = v_{exit}$$

$$h = h_{in} - v^2/2$$

$$p = p_{in} + (p_t - p_{in}) P_{rec}$$

where v_{exit} is some specified exit velocity, P_{rec} is a specified pressure recovery coefficient, and the subscript in corresponds to the inlet values. A call to prop then gives the exit values for the flow temperature, entropy, and density. The exit flow area and diffuser area ratio are calculated as

$$A_{ex} = \frac{m}{\rho v}$$

$$A_{ratio} = \frac{A_{ex}}{A_{in}}$$

In the off-design mode, the pressure recovery coefficient is still required; however, the exit velocity is determined based on a specified exit area. Thus, in the above equations, v_{exit} is iterated until the calculated exit area is the specified value.

The differser model parameters are defined as follows:

mode -	Character string specifying either design,"d", or off-design,"o", modes ("d"). Input.
vel -	Exit velocity (10.0 m/s) from the diffuser. Input.
pres_rec -	pressure recovery coefficient (0.5). Input.
ain -	Inlet flow area calculated based on inlet mass flow rate and inlet velocity. Output.
aex -	Exit flow area. Output in the design mode and input in the off-design mode (1 m^2).
aratio -	Exit area divided by inlet area. Output.
fl -	Exit flow from the model. Output.

5.3.12 Nozzle (nz) Model Class

The nozzle (`nz`) model class represents a gaseous flow nozzle. The model has one calculational member function, `c`, taking one input flow from the `gass` stack and putting one output flow back on the `gass` stack.

The model has both design and off-design modes. In the design mode, the model makes use of a specified exit pressure p_{exit} and a call to the `prop` function with the inlet entropy value to determine the enthalpy h_s for an isentropic expansion to the exit pressure. The exit flow velocity is then determined from

$$v = \sqrt{v_{in}^2 + 2\eta(h_{in} - h_s)}$$

where the `in` subscript denotes the inlet conditions and η is a specified nozzle efficiency. The exit flow enthalpy is then found from

$$h = h_{in} + (v_{in}^2 - v^2)/2$$

with the rest of the variables for the exit flow determined by a call to prop with the exit enthalpy as input. The exit area is then calculated from

$$A = \frac{m}{\rho v}$$

For use as output variables the exit Mach number, thrust, and specific impulse are then calculated from

$$\mathrm{Mach} = v\sqrt{\left(\frac{\partial\rho}{\partial p}\right)_s}$$

$$\mathrm{thrust} = mv + pA$$

$$\mathrm{impulse} = (\mathrm{thrust})/(9.8m)$$

where m is the mass flow rate, A is the exit flow area, and $\left(\frac{\partial\rho}{\partial p}\right)_s$ is calculated via finite differences using calls to `prop`.

In the off-design mode, the efficiency is still input, but the exit pressure is now iterated over until the calculated area is equal to the specified exit area.

The nozzle model parameters are defined as follows:

mode -	Character string representing, "d", design mode or "o", off-design mode ("d"). Input.
pres -	Exit pressure of the nozzle (0.5 atm). Input.
eff -	Efficiency of the nozzle (0.85). Input.

areain -	Inlet flow area (m^2) of the nozzle. Output.
area -	Exit flow area (m^2) from the nozzle. Input in the off-design mode, output in the design mode.
mach -	Exit mach number form the nozzle.
thrust -	Thrust (N) generated by the nozzle.
impulse -	Specific impulse (s) of the nozzle.
fl -	Exit flow from the model.

5.3.13 Steam Drum (sd) Model Class

The steam drum (sd) model class represents a conventional steam/water separator. The model makes use of two member functions, c and s. The c function takes one gastype flow from gass, representing the entering two-phase flow and, on exit from the model, puts one gastype flow representing the liquid phase flow back on the stack. The s function, which should only be called after the c function, requires no input flows and generates one gastype flow representing the exit steam flow. Note that while the model is called a steam drum, it will also handle fluids other than steam/water. It should not be called, however, with an input flow having an id of "GAS".

The modeling within sd consists of first calling the sat property code to determine the liquid and vapor saturation enthalpies, h_l and h_v, at the model's inlet. If the entering pressure is greater than the critical pressure, then a message is printed and the run is terminated.

The steam mass is calculated on the basis of the inlet flow's quality, q_{in}. If q_{in} is less than zero, then the steam mass, m_s, is zero. If q_{in} is greater than one, then the steam mass is set equal to the inlet flow's mass, m_{in}. If q_{in} is between 0 and 1, then the steam mass is calculated as

$$m_s = q_{in} m_{in} .$$

In each case the mass of the exiting liquid phase, m_l, is calculated from

$$m_l = m_{in} - m_s .$$

The exit enthalpies for the liquid and steam flows are set to the liquid and vapor saturation enthalpies, respectively, then the prop code is called to determine the entropies, densities, etc., for both flows.

The steam drum model parameters are defined as:

fl -	exit liquid flow from the model.
fls -	exit steam (vapor) flow from the model.

5.3.14 Feed Water Heater (fh) Model Class

The feed water heater (fh) model handles a conventional feed water heater; and, like the steam drum model, it will handle fluids other than steam/water. This model makes use of three member functions, s, h, and c. The h function represents the hot side flow, usually an extraction flow from a turbine. This function takes one flow from the gass and puts back one flow representing the flow from the drain cooler. The s function is used to pick up any cascade flow from higher pressure feed water heaters. This function takes one flow from the gass, but generates no output flows. The s function is only called if the feed water heater takes a cascade flow but, if used, must be called before the h entry. The c entry represents the cold feed water flow and takes one flow from the gass and generates one flow on exit. This entry must only be called after the h entry has been called.

On entry to the h function, a check is made to see if the entering flow is supercritical (i.e., inlet pressure greater than critical pressure). The model expects the hot side flow to be subcritical. If it is not, a message is printed and the run is terminated. The sat property code is then called to determine the saturation liquid and vapor enthalpies, h_l and h_v. If the entering flow's enthalpy is greater than h_v the flow is cooled to the saturation line with the amount of cooling necessary calculated from

$$q_0 = m_h(h_h - h_v)$$

where h_h is the enthalpy of the hot steam flow, and m_h is its mass flow rate.

When used, the s function will set a flag, cascade, indicating that a cascade flow exists. This flag is checked, and if a cascade flow exists, it is mixed with the hot steam flow. This is done by recalculating the mass flow rate and enthalpy on the hot side, as follows:

$$h_h = \frac{m_{cas} h_{cas} + m_h h_h}{m_{cas} + m_h}$$

$$m_h = m_{cas} + m_h$$

Here the subscript cas refers to the cascade flow. At this point, if h_h is greater than h_l, the combined flow is cooled to the liquid saturation line with the amount of cooling calculated from

$$q_1 = m_h (h_h - h_l)$$

The prop code is then called to determine the flow properties at this point.

If any subcooling has been specified in the feed water (such as in a drain cooler section), the hot flow temperature, t_h, is then reduced by the amount of subcooling desired. The prop code is then called to determine the new flow properties after subcooling, and the cooling load during subcooling is calculated from

$$q_2 = m_h (h_{before} - h_h)$$

where h_{before} is the enthalpy of the flow before subcooling. From a thermodynamic point of view, one could simplify the above by calculating the amount of cooling to the saturation liquid line or to the degree of subcooling, directly. However, the three cooling loads, q_0, q_1, and q_2, are calculated individually, as they correspond to the de-superheating, condensing, and drain cooler regions of the feed water heater.

In the cold-side function, c, the three values of q_0, q_1, and q_2 are used to heat up the feed water. This is done by redefining the cold-side enthalpy, h_c, through the three regions using

$$h_c = h_c + \frac{q_i}{m_c} \qquad i=0,\ 1,\ 2$$

where m_c is the mass flow rate on the cold side. The prop code is called after each region to determine the flow's state point. On both the c and h sides, the temperatures of the flows at the exit of each region are stored for later printout.

The model parameters for the feed water heater are defined as:

subcool -	Degree of subcooling desired within the drain cooler region (10 K). Input.
cascade -	Flag indicating whether or not the s entry has been called.
q[3] -	Array of heat loads in the three regions of the feed water heater (w). Output.
htemp[4] -	Array of hot-side temperatures at the inlet and exits of the three regions of the feed water heater (K). Output.
ctemp[4] -	Array of cold-side temperatures at the inlet and exits of the three regions of the feed water heater (K). Output.
flh -	Exit hot-side flow from the feed water heater or drain cooler.
fls -	Inlet cascade flow.
flc -	Exit cold-side flow from the feed water heater.

5.3.15 Combustor (cb) Model Class

The combustor (cb) model class is used to simulate the burning of a fuel with an oxidizing gas flow. The fuel is described by the input parameters of the model while the oxidizing flow is taken from the gass stack, and must be a flow with a "GAS" id. The model has one calculational member function denoted as c, which takes one input flow from the gass stack and puts one output flow back on the stack.

On entry to the model a reference gas calculation is made to determine the heat of formation of the fuel. This is done by first calculating the mass flow rate of oxygen necessary to burn the fuel at a stoichiometry of one from

$$m_o = (2.6641w_c + 7.93645w_h + 0.99797w_s - w_o)m_{fuel}$$

where w_c, w_h, w_s, and w_o are the weight fractions of carbon, hydrogen, sulfur, and oxygen in the fuel, and m_{fuel} is the mass flow rate of the fuel. The molar flow rates for the carbon, hydrogen, sulfur, water, nitrogen, and oxygen (including m_o) for a reference gas consisting of the fuel plus oxidizer (at stoichiometry of one) are then determined from

$$mol_i = w_i(m_{fuel}/mw_i)$$

where the subscript i stands for either carbon, hydrogen, water, sulfur, nitrogen, or oxygen, and mw_i is the corresponding molecular weight for these species. By calling the prop function for this reference combustion gas at a temperature of 298.16 K and a pressure of 1.0 atm, and preventing any condensed phases from forming using the gass noform array, a reference enthalpy as well as the equilibrium composition can be determined. If the lower heating value, LHV, of the fuel is known, the heat of formation of the fuel (Δh_{form}) can be determined from

$$\Delta h_{form} = ((m_{fuel} + m_o)h + m_{fuel}LHV)/m_{fuel}$$

where h is the reference enthalpy calculated above.

Once this reference gas calculation is done, the actual oxidizing flow can be used to determine the stoichiometry of the combustion from

$$stoich = \frac{m_{ox}}{mw_{ox}} \frac{31.9988}{m_o} comp_{o2}$$

where $comp_{O2}$ is the mole fraction of O_2 in the oxidizing flow, m_{ox} is the mass flow rate of the oxidizing flow, and mw_{ox} is its molecular weight. The molar flow rates of the actual combustion gas species consisting of the original fuel species and the actual oxidizing flow species can be determined from

$$mol_i = \frac{m_{ox}}{mw_{ox}} comp_i + (mol_i)_{fuel}$$

Here $(mol_i)_{fuel}$ is the same molar flow rate as for the reference gas but without the m_o moles of oxygen. The molar rates can be normalized to yield the mole fractions for the combustion gas species and, through a call to the atom function, the atom fractions for the combustion gas. The enthalpy and mass flow rate of this gas are then determined from

$$h = \frac{m_{ox}h_{ox} + m_{fuel}\Delta h_{form}}{m_{ox} + m_{fuel}}$$

$$m = m_{ox} + m_{fuel}$$

A call to the prop function with this enthalpy as the input (and at the pressure of the oxidizing flow) will then give the flame temperature of the combustion products, as well as their equilibrium composition and other state variables.

For use in power summaries, the input power to the combustor is stored as

$$Pow = m_{fuel}LHV$$

The combustor model parameters are defined as follows:

mass -	The mass flow rate of the fuel (1.0 kg/s). Input.
carb -	The carbon weight fraction within the fuel (0.25). Input.
h -	The hydrogen weight fraction within the fuel (0.75). Input.
o -	The oxygen weight fraction within the fuel (0.0). Input.
s -	The sulfur weight fraction within the fuel (0.0). Input.
n -	The nitrogen weight fraction within the fuel (0.0). Input.
h2o -	The water weight fraction within the fuel (0.0). Input.

lhv -	The lower heating value (J/kg) of the fuel (10^7). Input.
stoich -	Ratio of oxygen within the oxidizer to the amount of oxygen necessary for 100% fuel oxidation.
power.heat -	Total thermal power input equal to lhv times the fuel mass.
fl -	Combustion gas flow from the model.

The combustor model should only be used with oxidizing flows having the "GAS" id. Note that this model restricts the species that must be included with the gas properties code, that is, the following species must be included: C, CO, CO_2, H_2, H_2O, S, SO_2, and N_2 must be included.

5.3.16 Condenser (cond) Model Class

The condenser (cond) model represents the condensation of water from a hot gas flow. The model has two calculational entries. The first, denoted as c, takes one gastype input flow with a "GAS" id and generates one gastype output flow. This entry processes the hot gas flow and must be called before the secondary, s entry, which generates one output gastype flow, with a "STM" id representing the condensed water flow. A third entry, cool, which also must only be called after the c entry, is used to represent a coolant fluid flow and takes one gastype flow from the gass stack and puts one gastype flow back onto the stack.

On entering the c function, the molar flow rates of condensed water and water vapor within the inlet flow are saved as $n_{in,h2o,cond}$ and $n_{in,h2o}$. The gas flow is then brought down to the specified exit temperature, and the prop code is called to determine the new equilibrium composition. The molar flow rate of water that is condensed from the entering water vapor during this temperature change is then calculated from

$$n_{h2o,cond} = n_{in,h2o} - n_{ex,h2o}$$

where $n_{ex,h2o}$ is the new equilibrium molar flow rate of water vapor at the exit. If $n_{h2o,cond}$ is less than zero, it is set to zero. The total molar flow rate of condensed water removed from the flow is then calculated as

$$n_{cond,tot} = n_{in,h2o,cond} + n_{h2o,cond}$$

The mass flow rate of the entering gas flow is reduced by this amount of condensed water,

$$m = m_{in} - n_{cond,tot} mw_{h2o}$$

where m_{in} is the inlet flow rate of gas. Note that this procedure removes any inlet condensed water even if the exit temperature were to increase and, theoretically, vaporize it. A secondary flow is then created with the property id of "STM", a flow rate equal to the mass flow removed from the entering gas stream, and a temperature and pressure of the exiting gas stream. The prop code is called for this new flow, which is saved for use in the s entry to the model. The heat removed from the gas stream is calculated as the change in enthalpy from the inlet temperature to the new exit temperature,

$$Q = m_{in}(h_{in} - h_{ex})$$

Within the cool entry the coolant flow's exit enthalpy is adjusted from its input value by the addition of the Q heat

$$h_{cool,ex} = h_{cool,in} + \frac{Q}{m_{cool}}$$

The property code then gives the exit temperature of the coolant flow. For use in sizing the condenser, a specified input overall heat transfer rate, u(w/m^{2}K), is used along with a calculated log-mean-temperature difference, ΔT, to give an area, A, from

$$A = \frac{Q}{u\Delta T}$$

As with the heater and heat exchanger models, a rough condenser weight is calculated from

$$W = 0.5AT\rho$$

where W is the weight, T is the wall thickness, and ρ is the wall material density.

The condenser model parameters are defined as follows:

texit -	Exit temperature of the input gas flow (323 K). Input.
heat -	Total heat removed from the input gas flow in cooling the flow to the exit temperature (W). Output.
pvap -	Vapor pressure of water at the exit temperature (atm). Output.
ph2o -	Partial pressure of water in the gas flow at the exit (atm). Output.
h2ocond -	Moles/s of water condensed from the gas flow. Output.
h2oout -	Moles/s of water vapor leaving with the gas flow. Output.
h2oin -	Moles/s of water vapor entering with the gas flow. Output.
u -	Overall heat transfer coefficient (W/m^{2}K) of the heat exchange process (300). Input.
area -	Total heat transfer surface area (m^2). Output.
lmtd -	Log mean temperature difference (K) of the heat transfer process. Output.
thick -	Wall thickness (0.001 m). Input.
dens -	Wall material density (7800 kg/m^3). Input.
fl -	Main gas flow. Output.
fls -	Condensate water. Output.
flcool -	Coolant flow. Output.

5.3.17 Dynamic Heater (dht) Model Class

The dynamic heater/cooler (dht) model is a multi-nodal version of the heater/cooler (ht) model. The model is very similar to the multi-nodal option of the pipe model, only here, heat exchanges with the wall are permitted. The model has one calculational function denoted as c, which requires one gastype input flow and produces one gastype exit flow.

On entry to the model, the pressure drop across the heater is determined by using the same friction factor correlation as used within the pipe model. In doing this, the flow is divided into a number of equal flow passages, npass, each of the same flow height and width. The hydraulic diameter of each passage is then calculated as 4 times the flow area divided by the perimeter of the flow. As with the pipe model, the total pressure drop is calculated based on the heater's length, with each node taking 1/nnode times the total pressure drop. The exit enthalpy of each node is calculated from

$$\rho V\frac{\partial h_i}{\partial t} = \dot{m}(h_{i-1} - h_i) + uA(Tw_i - T_i)$$

where ρ is the fluid density, V is the node volume per flow passage, $\dot{m}$ is the flow rate per flow passage, h_{i-1} and h_i are the inlet and exit node enthalpies, u is the overall heat transfer coefficient, A is the heat transfer surface area per node per passage, Tw_i is the wall temperature at node i, and T_i is the fluid temperature at node i. The wall temperatures are determined from

$$cp_{wall}M_{wall}\frac{\partial}{\partial t}Tw_i = Q - uA(Tw_i - T_i)$$

where cp_{wall} is the specific heat of the wall material, M_{wall} is the mass of a wall node per flow passage, and Q is a specified heat input per node per flow passage.

For the initial entry to the model, the above equations along with the pressure drop equations are solved with the time derivatives set to zero to determine the initial values of T_{wi} and h_i.

The dynamic heater model parameters are defined as follows:

heat -	Total heat transferred to the wall (10^5 W). Input. Note that heat can be set to a negative number to define a flow cooler.
dh -	Enthalpy change of the fluid across the heater (W). Output.

surfarea -	Total heat transfer surface area (m^2). Output.
u -	Overall heat transfer coefficient (30 W/m^2K). Input.
length -	Length of heater (1 m). Input.
width -	Width of heater (0.5 m). Input.
heightpass -	height of a flow passage (0.015 m). Input.
volpass -	Volume of a flow passage (m^3). Output.
thickwall -	Thickness of the wall material (0.001 m). Input.
denswall -	Density of the wall material (7800 kg/m^3). Input.
masswall -	Mass of the wall per flow passage (kg). Output.
cpwall -	Specific heat of the wall material (600 J/kg-K). Input.
re -	Inlet flow Reynold's number. Output.
fric -	Friction factor. Output.
mu -	Fluid viscosity. Output.
muref -	Fluid viscosity at reference temperature ($2.671x10^{-5}$). Input.
tref -	Refence temperature for visocity expression (500 K). Input.
nref -	Exponent in the visocity expression (0.6364). Input.
rough -	Sand grain roughness/hydraulic diameter for the pipe (0.0). Input.
nnode -	Number of nodes used (5). Input. Nnode should be less than or equal to 9.
h[10] -	Array of node exit enthalpies. Output.
t[10] -	Array of node exit temperatures. Output.
twall[10] -	Array of node wall temperatures. Output.
fl -	Exit flow from the model. Output.

5.3.18 Dynamic Heat Exchanger (dhx) Model Class

The dynamic heat exchanger (`dhx`) model is a dynamic version of the heat exchanger (`hx`) model. Unlike the dynamic heater model, where multi-nodes are used the `dhx` model only makes use of a single node, but also makes use of log mean temperature differences when the heat transfers between the walls and fluid flows are being calculated. This was done to provide more accuracy when a counter flow device is being handled. A large number of nodes would be required to accurately reflect the temperature distributions along a counter flow device. Note that while log mean temperature differences might be used for each node of a multi-nodal technique, one often runs into stability problems as well as the problems of defining a log mean temperature when the temperatures cross over. For these reasons and for execution speed only a single node has been used in `dhx`. Log mean temperature differences, of course, really only make sense when one is not in a two-phase region where the specific heats become infinite. However, in these regions the limiting heat transfer is usually on the side of the single-phase heat transfer. Thus, provided one has at least one side of the heat exchanger in single phase, the use of the log mean temperature differences is probably sufficient for most system calculations.

The dynamic heat exchanger model is set up slightly differently from the `hx` model in that both the hot and cold inlet flows must be known before the model can calculate either of the exit flows. This makes the model a little more difficult to use than the `hx` model. The `dhx` model has four member functions: `cin`, processing the cold inlet flow; `hin`, processing the hot inlet flow; `cout`, processing the cold outlet flow; and `hout`, processing the hot side outlet flow. The two inlet entries require one `gastype` input flow and generate no output flows, while the two outlet entries require no input flows but generate one `gastype` output flow. The two inlet entries must be called before either one of the outlet entries is called. It does not matter which outlet entry is called first.

Once both inlet flows are known, the model will calculate the state points for the exit fluid flow and the wall temperature at each end of the heat exchanger. The differential equations used to represent the conservation of energy within the fluid flows are as follows:

$$\rho_c V \frac{\partial h_{c1}}{\partial t} = \dot{m}_c (h_{c0} - h_{c1}) + u_c A \Delta T_{wc}$$

$$\rho_h V \frac{\partial h_{h1}}{\partial t} = \dot{m}_h (h_{h0} - h_{h1}) - u_h A \Delta T_{hw}$$

In these equations, the subscripts c and h refer to the cold and hot flows, respectively; subscript w refers to the wall, subscripts 0 and 1 refer to the inlet and exit conditions; ρ is the density; V is the fluid flow volume; h is the fluid enthalpy; m is the mass flow rate; ΔT is the log mean temperature difference; u is the effective heat transfer coefficient; A is the heat transfer surface area; and t is the time. Both V and A are taken to be equal for both fluids and, thus, do not need to have subscripts. These two equations represent the dynamic enthalpy balance. The first term (left-hand side) represents the change in the total enthalpy within the fluid volume, which is balanced by the enthalpy flow into and out of the fluid volume (first term on the right-hand side) and the heat transfers into or out of the fluid volume.

For the wall temperatures at the two ends, T_{w0} and T_{w1}, the following equations are used to represent the energy conservation:

$$\frac{M_w}{2} Cp_w \frac{\partial T_{w0}}{\partial t} = u_h A \Delta T_{hw} f_{h0} - u_c A \Delta T_{wc} f_{c0}$$

$$\frac{M_w}{2} Cp_w \frac{\partial T_{w1}}{\partial t} = u_h A \Delta T_{hw} f_{h1} - u_c A \Delta T_{wc} f_{c1}$$

Here M_w is the total mass of the wall material, Cp_w is the specific heats of the wall, and the f's are discussed below. These equations express the rate of change of the end-point wall temperatures as a function of the local heat transfer into or out of the walls due to the fluid flows.

The above equations have been described with the fluid flows representing a co-flow heat exchanger. In the case of a counter-flow exchanger, the subscripts 0 and 1 representing the two ends are adjusted accordingly.

In each of the equations, the log mean temperature difference is dependent on both end-point temperatures of one of the fluid flows and the wall. For the normal case of the hot flow being hotter than the wall at both ends, the log mean temperature is defined as

$$\Delta T_{hw} = \frac{(T_{h0} - T_{w0}) - (T_{h1} - T_{w1})}{\log \frac{(T_{h0} - T_{w0})}{(T_{h1} - T_{w1})}}$$

This cannot be used, however, for a temperature crossover situation which can occur in a dynamic situation due to the changing inlet flow temperature. In such a situation the actual heat transfer along the fluid flow path is both positive and negative. However, such situations are generally transients, and thus ΔT_{hw} is adjusted to approximately represent the driving temperature and to qualitatively represent the way the temperature fields would change in such a crossover situation.

The adjustment to the expression for the log mean temperature difference is given by the following algorithm. Let x be the temperature difference between the entering flow and wall, and y be the same temperature difference at the exit. Then, ΔT is defined through the following equations.

$$s = \mathrm{sign}(y)$$

$$r = \min\left(\frac{\max(|x|, |y|)}{\min(|x|, |y|)}, 1000\right)$$

For r=1,

$$\Delta T = s|x|$$

Otherwise,

$$\Delta T = s \min(|x|, |y|) \frac{(r-1)}{\log(r)}$$

Note that the sign of ΔT is defined by the temperature difference at the exit. For the hot flow, a positive inlet and a negative exit temperature difference would make ΔT negative, driving the heat transfer negative (i.e., from the wall to the hot flow). Thus, the exit flow temperature would increase and/or the wall temperature would decrease until the crossover condition is eliminated. A negative inlet temperature difference (wall hotter than the hot gas) and a positive exit temperature difference would again drive the flow and wall temperatures closer together. The same expressions can be used for the driving temperature difference between the wall and the cold flow. Note that by factoring out the minimum value of $|x|$ or $|y|$, the above ΔT expression is linearized as a function of the minimum temperature difference in regions where there is a pinching down of the flow and wall temperatures. In most cases, the above algorithm gives the usual log mean temperature difference.

Along with this adjustment to the log mean temperature difference, an adjustment is also made as to how the heat is distributed to or from the two wall nodes. This is accomplished by the f factors in the wall energy equations. Normally, when no temperature crossover occurs, the factors are defined as follows:

$$f_{h0} = \frac{|x|}{|x| + |y|}$$

$$f_{h1} = 1 - f_{h0}$$

with f_{c0} and f_{c1} defined similarly. This gives a partitioning of the heat transfers proportional to the temperature difference. However, in a temperature crossover situation at the inlet (i.e., $T_w > T_h$), the f_{h0} and f_{h1} factors become 0 and 1, respectively, causing the heat transfer to shift toward the exit region. This is done to more closely represent the actual heat transfers in these transient conditions. When a crossover situation occurs, the flow is both heated and cooled by the wall in such a way as to drive the exit temperatures closer together. This requires that the sign of the net heat transfer be the same as that of the exit temperature difference. However, unless the f factors are readjusted as above, this would tend to drive the inlet temperatures further apart. The f_c factors are adjusted similarly when the cold inlet flow exceeds that of the wall temperatures. In should be reiterated that the adjustments to these f factors are only done during a temperature crossover.

At the start of the simulation, the model offers three options for setting the wall temperatures. The first option is related to the design mode. In this option, an input value for exit flow temperature or exit flow quality (for condensable flows)on the hot or cold side is specified. Assuming steady-state conditions at the design point and using the known inlet flow conditions, the overall heat transfer, Q, across the heat exchanger and exit flow conditions can then be determined. The wall temperatures are then determined from the following steady-state values:

$$T_{w0} = \frac{u_h T_{h0} + u_c T_{c0}}{u_h + u_c}$$

$$T_{w1} = \frac{u_h T_{h1} + u_c T_{c1}}{u_h + u_c}$$

The heat transfer surface area can then be determined using the log mean temperature difference on either the hot or cold side and the overall heat transfer:

$$A = \frac{Q}{u_c \Delta T_{wc}}$$

The second option is an off-design mode. Again steady-state conditions are assumed, only now the overall heat transfer, Q, is varied until the surface area calculated by the above equations is equal to a specified value.

Finally, the third option simply makes use of a specified input set of wall temperatures and a specified surface area. In this case it is not assumed that the overall heat exchanger is in steady-state conditions, but that the hot flow and the cold flow each satisfy

$$A = \frac{Q_h}{u_h \Delta T_{hw}}$$

$$A = \frac{Q_c}{u_c \Delta T_{wc}}$$

where Q_h and Q_c are individually varied to satisfy these equations. Thus, the heat transferred to/from the walls on either side of the exchanger will not necessarily be equal.

Once the heat exchanger's area is known, the exchanger's length, L, is determined by using an input value for the exchanger's width, W, and the number of parallel flow passages, N, from

$$L = \frac{A}{2WN}$$

The volume of the fluid flow passages is then determined from

$$V = NWH_p$$

where H_p is the height of a single flow passage. Finally, the total wall mass is determined from the surface area and specified input values of the heat exchanger's wall thickness, H_w, and the density, ρ_w:

$$M = AH_w\rho_w$$

The model will work properly if the flow passage height is set to zero. In this case, the time derivative terms are set to zero in calculating the exit flow conditions, that is, the flows are treated in a quasi-steady-state way. The wall temperature equations, however, are solved just as before.

The parameters for the dynamic heat exchanger model are as follows:

npass -	Number of passages through the heat exchanger (10). Input. Note that a single passage corresponds to one hot-side and one cold-side flow.
prt -	Print flag for generating intermediate results (0). Input.
width -	Width of a flow passage (0.5 m). Input. Width is also the width of the entire heat exchanger since all flow passages are of equal width.
heightpass -	Height of a flow passage (0.0 m). Input. This is the height of either the hot- or cold-side flow passage, as both are taken as equal. A zero value treats the flows as being quasi-steady state.
length -	Length of a flow passage and of the entire heat exchanger (m). Output.
surfarea -	Total heat transfer surface area (m^2). Output for the design mode or input for the off-design mode.
volpass -	Volume of the hot- or cold-side flow passages (m^3). Output. This is the volume of all the flow passages.
pf_cold -	Fraction of the inlet pressure used as a pressure drop through the cold-side passages (0.0). Input.
pf_hot -	Fraction of the inlet hot flow pressure used as a pressure drop through the device (0.0). Input.
denswall -	Density of the wall material (7800 kg/m^3). Input.
cpwall -	Specific heat of the wall material (600 J/kg-K). Input.
thickwall -	Thickness of the walls (0.001 m). Input.
masswall -	Total mass of the wall material (kg). Output.
ufc -	Film heat transfer coefficient on the cold side (30 W/m^{2}K). Input.
uhc -	Film heat transfer coefficient on the hot side (30 W/m^{2}K). Input.
t_cold -	Initial cold side exit temperature (0.0 K). Input.
t_hot -	Initial hot side exit temperature (0.0 K). Input.
q_cold -	Initial cold side exit quality (-1000). Input.
q_hot -	Initial hot side exit quality (-1000). Input.

twall[2] -	Initial wall temperatures at cold flow inlet and exit (300.0 K). Input when the mode is set to "t".
tc[2] -	Cold-side flow temperatures at the cold-side inlet and exit, respectively (K).
th[2] -	Hot-side flow temperatures at the cold-side inlet and exit locations, respectively (K).
dhh -	Total mass flow rate times enthalpy change from inlet to exit for the hot side flow (W).
dhc -	Total mass flow rate times enthalpy change from inlet to exit for the cold side flow (W).
lmtd -	Overall hot-flow to cold-flow log mean temperature difference (K).
mode[2] -	Character string taking the values of either "d", "o", or "t" for design, off-design, or specified wall temperature mode ("d"). Input.
type[8] -	Character string taking the values of "count" for a counter flow heat exchanger or "paral" for a parallel flow heat exchanger ("count"). Input.

In the design mode, the `dhx` model is similar to the `hx` model in that one of either `t_cold`, `t_hot`, `q_cold`, or `q_hot` needs to be specified. Thus, only one of these values should be specified, the others should be left at their default values.

5.3.19 Reformer (reform) Model Class

The reformer (`reform`) model is used to simulate a hydrocarbon fuel reformer. The model has four functional entries. The first (`s`) and is used to obtain an input water flow for steam reforming or an input oxidizing flow for partial oxidation reforming. This entry must be called before the main calculational entry, `c`, and requires one `gastype` flow on input and generates no output flows. This flow may have a property `id` of either "GAS", "STM", or "THR-species". The second entry is denoted as `a` and can optionally be called to pick up an additional flow to be mixed with that within the `s` entry. This entry must also be called before the `c` entry and requires one `gastype` flow on input but generates no output flows. The flow in the `a` entry may also have a property id of either "GAS", "STM", or "THR-species". This `a` entry might be used, for example, when both steam and air are combined within the reformer. The third entry, `c`, represents the main calculational entry and requires one `gastype` flow representing the fuel as input and generates one `gastype` flow on output. The input fuel flow may have either the "GAS", "STM", or "THR-species" `id` and will have the "GAS" `id` on output. In all of these entries, for the "STM" or "THR-species" type flows, the `comp` array needs to be defined (consistent with the flow) so that a "GAS" type flow can be generated. The fourth entry, denoted as `h`, is used to represent any hot-side burner gas flow. This entry requires one `gastype` flow on input and generates one `gastype` flow on output. Entry `h`, which is optional, should only be called after the `c` entry has been called.

The modeling within the main calculational entry is very similar to that of a mixer. First, the input flows are converted to "GAS" type flows and their input enthalpies saved. The resulting flows are then mixed together, and the equilibrium chemical composition of the gas is calculated either at a specified exit temperature or at an enthalpy equal to the sum of the inlet enthalpies, depending on the input option. The total heat required by the process is calculated on the basis of the total enthalpy change of the entering and leaving flows. This heat is then extracted from the burner gas flow in the `h` entry. Note that this entry is only required when an exit temperature of the primary flow is specified; otherwise, heat is zero. For sizing purposes, the heat transfer surface area times the effective heat transfer coefficient is also calculated within the `h` entry, based on whether the flow configuration is either counter flow or parallel flow. For use in this calculation, the inlet temperature for the combined fuel, air, and water flows is calculated as a mass-weighted value of the three input temperatures.

The reform model parameters are as follows:

texit -	Specified exit temperature of the reformed gas (650 K). Input.
heat -	Total heat required by the reforming process (W). Output.
option -	Character string specifying either "temp" or "enth", indicating that either a specified exit temperature has been input, or that the exit temperature is the result of the sum of the inlet flow enthalpies ("temp"). Input.
type -	Character string taking the values of either "count" or "paral", indicating the type of flow configuration between the hot side and the reformate side ("paral"). Input.

lmtd -	Log mean temperature difference across the device (K). Output when the h entry is called.
tmix -	Mass-weighted average of the inlet flow temperatures (K). Output.
ua -	Heat transfer coefficient times the heat transfer surface area. Output.
fl -	Reformed gas output at exit.
flh -	Burner gas output at exit.
fls -	Water or oxidizing flow at inlet.
fla -	Additional water or oxidizing flow at inlet.

5.3.20 Generic Dynamic Flow Reactor (reac) Model Class

The generic dynamic flow reactor (`reac`) model simulates a one-dimensional, time-dependent flow reactor. The model can be used to represent fuel reformers, shift converters, preferential oxidizers, or other devices in which a flow undergoes kinetic reactions along the flow direction. Since flow reactors usually represent some catalytic process, various options can be specified by the user to define the reaction rates along the device.

The model permits up to three input flows obtained by optionally calling either an `s` or `a` entry, followed by the main calculational entry, `c`, similar to the reform model. Each of these entries requires one `gastype` flow on input. For the `s` and `a` entries, no output flows are generated, and for the `c` entry, one `gastype` flow is generated, which represents the exiting reacted flow stream. Each of the input flows may have a property id of either "GAS", "STM", or "THR-species". For the "STM" or "THR-species" type flows, the `comp` array needs to be defined (consistent with the flow) so that a "GAS" type flow can be generated. Normally, the `s` entry can be thought of as a steam flow and the `a` entry as an air or oxidizer flow. Since this model makes use of reaction rates to calculate the speciation of the flow, the flow to the main calculational entry should generally be frozen. This will be discussed further below.

The modeling within `reac` proceeds as follows. On entry to the model the input flows are combined into a single flow with the combined molar flow rates of the species from all flows as determined from the `comp` arrays. Additionally, the enthalpy of the combined flow is determined as the sum of the mass-weighted enthalpies of each flow. An inlet pressure level is then determined on the bases of the lowest of the entering flow pressures. At present, the pressure along the device then drops linearly to a total pressure drop specified by the user.

For this simple model of a reactor, the approximation is made that the time scale of interest is substantially larger than the time scale of pressure or density fluctuations, which would propagate through the device with sonic velocity. Because of the very large heat transfer rates between a porous catalyst bed and the surrounding gases, it is assumed that the temperature of the bed material is essentially the same as that of the gas. It is also assumed that the device is sufficiently insulated that no heat is conducted to its surroundings. Thus, the equations of conservation of mass, species concentrations, and energy are defined as

$$\varepsilon\frac{\partial\rho_g}{\partial t} + \varepsilon u_g\frac{\partial\rho_g}{\partial x} = 0$$

$$\varepsilon\frac{\partial C_i}{\partial t} + \varepsilon u_g\frac{\partial C_i}{\partial x} = S_i$$

$$\varepsilon\rho_g\frac{\partial h_g}{\partial t} + (1-\varepsilon)\rho_b c_{pb}\frac{\partial T_g}{\partial t} + \varepsilon\rho_g u_g\frac{\partial h_g}{\partial x} = 0$$

where ε is the porosity of the reactor bed material, ρ_g is the gas density (kg/m^3), ρ_b is the catalyst bed density (kg/m^3), u_g is the gas velocity (m/s), h_g is the gas enthalpy (J/kg), C_i is the molar concentrations of species i within the flow (mol/m^3), S_i is the source terms of species i (mol/m^{3}s), T_g is the gas temperature (K), and x and t are the axial location along the reactor and time, respectively. The gas enthalpy is calculated on the basis of reference states of the heat of formation of the individual species; thus, heats of reactions due to the reactions occurring do not explicitly appear within the conservation of energy equation. Note that the second of the above equations, when multiplied by the molecular weights (M_i) of the i species and then summed over all species, gives the first equation since the source terms must satisfy the consistency condition:

$$\sum_i M_i S_i = 0$$

Thus, the first of the above equations is not actually used within the code. Instead, a steady-state form of the mass conservation equation

$$\varepsilon \rho_g u_g A = \dot{m}$$

is used, where A is the cross-sectional area of the reactor, and $\dot{m}$ is the inlet mass flow rate to the reactor. This equation is then solved to give the gas velocity at each point along the device. Note that this assumes the inlet flow is only gradually changing with time.

The S_i terms represent the chemical reactions taking place within the reactor and are optionally defined for a specific device type or by means of a generic kinetics approach to equilibrium. For the generic kinetics option, which is activated by setting the input parameter, `type`, equal to "generic", the S_i terms are defined as

$$S_i = K(C_i^{eq} - C_i)$$

where C_i^{eq} is the chemical equilibrium concentrations for the flow at the current temperature and pressure, and K is defined as

$$K = e^{E_0\left(\frac{1}{T_0} - \frac{1}{T_g}\right)}$$

where E_0 and T_0 are user-supplied inputs.

When S_i is specified for a device type, the `type` parameter is set to a built-in device name. At present, only one device type is defined. It is for a preferential oxidizer, in which the kinetic reactions rates are defined for the following reactions:

$$CO + 1/2O_2 \rightarrow CO_2$$

$$H_2 + 1/2O_2 \rightarrow H_2O$$

$$CH_3OH + 1/2O_2 \rightarrow CO_2 + 2H_2$$

$$CO_2 + H_2 \rightarrow CO + H_2O$$

The rates for these reactions are then suitably combined to produce the S_i terms. This option is activated by setting the input variable `type` to "prox".

To integrate the above conservation equations, each is formulated by using a simple finite difference in both x and t, with the time-step controlled outside of the model as it is in all dynamic GCtool models. For a given user-supplied reactor length and number of spacial nodes or control volumes, the conservation equations can be integrated over each control volume. This results in a set of algebraic equations for the gas temperature and species concentrations at each node. These are nonlinear equations and are solved using the default GCtool equation solver. At each iteration of this process, the `prop` code is called with new values of T_g and C_i to obtain the current gas density and enthalpy. From the steady-state form of the conservation of mass equation, the gas velocity is then calculated. The S_i terms are then evaluated based on the user-supplied value of `type`, and the residuals of the finite difference conservative equations are evaluated. The equation solver then varies T_g and C_i until the residuals go to zero. This is then repeated for each control volume along the reactor.

To start the integrations over time, initial conditions along the reactor must be known. These are obtained either by assigning a user-input temperature along the reactor and then assuming that the flow compositions are in chemical equilibrium at that temperature or by taking the temperature and composition along the reactor to be the same as the inlet flow conditions to the reactor at time equal to zero. The first option is activated by setting the input parameter `tinit` to the desired reactor temperature. If `tinit` is zero, then the code takes the second option using the inlet flow conditions.

Several other options are also provided within the model. If the model is used in a steady-state run, then the time derivatives in the conservation equations are set to zero. Additionally, a chemical equilibrium option is available in which the species compositions are determined by using a call to the `prop` code rather than being defined through a set of kinetic reactions. In this case, it is not the species concentration equations that are integrated along the device, but the atom concentration equations,

$$\frac{\partial b_i}{\partial t} + u_g \frac{\partial b_i}{\partial x} = 0$$

where

$$b_i = \sum_j a_{ij} C_j$$

and a_{ij} represents the stoichiometric coefficients of atom i in species j. This option, which essentially represents infinitely fast reaction rates, is activated by setting the `type` input parameter to "equil". Note that when this equilibrium option is in effect, the frozen parameter for the main input flow is automatically set to 0; otherwise, it is set to 1 to prevent the `prop` code from reestablishing chemical equilibrium. Also, when `type` is "equil", the combined inlet flows are brought to equilibrium before integratiion of the flows along the device.

When the `type` parameter is not "equil", the `prop` code is still called at the inlet with the combined input flows by using the combined inlet enthalpy, but whether or not chemical equilibrium is established for the combined flow depends on the state of the frozen parameter for the main flow. Generally, the main flow should be frozen for this model since it is performing the chemical reactions internally rather than through the `prop` code. At times it may be appropriate to bring the inlet flows to a state of equilibrium before the integrations along the reactor to simulate some pre-oxidation chamber. In this case, rather than changing the main flow's frozen parameter, the `reac` model has an `ignite` parameter that can be set to 1 to force the combined inlet flows to a state of equilibrium at the combined inlet enthalpy. Note that the exit flow from the model has its `frozen` parameter set to 0 or 1, depending on whether `type` is set to "equil" or something else.

The parameters for the generic dynamic flow reactor model are defined as follows:

diam -	Diameter of the reactor (0.2 m). Input.
length -	length of the reactor (1.0 m). Input.
por -	Porosity of the catalyst bed material (0.3). Input
dp -	total overall pressure drop along the reactor (0.0 atm). Input.
nnode -	Number of nodes or control volumes along the reactor (5). Input. Must be less than 10.
thickwall -	Thickness of the wall surrounding the reactor (0.001 m). Input. Used along with rwall (below) to determine the weight of the wall material. This parameter is not used in any other calculations.
rwall -	Density of the wall material (8000 kg/m^3). Input.
rbed -	Density of the catalyst bed material (1200 kg/m^3). Input.
cpbed -	Specific heat of the catalyst bed material (500 J/kg-K). Input.
ignite -	Flag to turn on the ignite option to bring inlet flows to chemical equilibrium (0). Input.
tinit -	Initial bed temperature along the reactor if greater than zero (0.0 K). Input.
type -	Reaction type used in the reactor ("generic"). Input. At present, type can take on the values "generic", "prox", or "equil".
e0 -	Parameter in the generic kinetics option (3000.0). Input if type is specified as "generic".
t0 -	Parameter in the generic kinetics option (700.0). Input if type is specified as "generic".

5.3.21 Proton Exchange Membrane (pem) Model Class

The proton exchange membrane (`pem`) model simulates a PEM fuel cell for direct energy conversion. The model has four entries: `ain`, `c`, `a`, and `cool`. The `ain` entry is used to obtain the input anode flow. This entry requires one `gastype` flow and generates no output flows. This entry must be called before the cathode, or `c` entry, is called. The c entry requires one input gastype flow and generates one exit flow, representing the exit cathode flow. The flow id in the ain and c entries must be "GAS". A third entry, a, can then be called to obtain the anode exit flow. This entry requires no input flows and generates one output gastype flow. Finally, a cool entry is used to process the coolant flow through the cell. This entry requires one gastype flow as input and generates one gastype flow as output. The model has both a design and an off-design mode.

On entering the c function, the inlet enthalpies for each of the anode and cathode flows are saved as

$$\tilde{h}_{a,in}(T_{a,in}) = m_a h_a$$

$$\tilde{h}_{c,in}(T_{c,in}) = m_c h_c$$

where m is the mass flow rate, and h is the specific enthalpy. The subscript a represents the anode flow, c the cathode flow, and in the inlet values. The flows are then brought to the input cell temperature value, T_{cell}, where the flow enthalpies and entropies are then calculated and saved as

$$h_{a,in}(T_{cell}) = m_a h_a(T_{cell})$$

$$s_{a,in}(t_{cell}) = m_a s_a(T_{cell})$$

where s is the specific entropy. A similar set of equations hold for the cathode flow. The molar flow rate for the anode flow is calculated from its molecular weight and the species mole fractions:

$$n_{a,in,i} = (x_{a,in,i} m_a)/(mw_a)$$

A similar equation is used for the cathode flow. Here, i is the species index, n is the molar flow rate, x is the species mole fraction, and mw is the flow's molecular weight.

The model has several modes for determining the amount of cell reaction that takes place. In the design mode, an input fuel (hydrogen) utilization, u_{h2}, is used to determine the total moles (per second) of hydrogen consumed by the cell:

$$n_{h2,cons} = u_{h2} n_{a,in,h2}$$

The total cell surface area is then determined from

$$A = (2n_{h2,cons}F)/i$$

where F is the Faraday constant (Coulombs/kg-mole), and i is an input cell current density. In the off-design mode, it is assumed that the cell area is input. The input cell current density can then be used to determine the total number of moles of hydrogen consumed by

$$n_{h2,cons} = (iA)/(2F)$$

and the fuel utilization from

$$u_{h2} = n_{h2,cons}/n_{a,in,h2}$$

A third mode, denoted as the utilization mode, makes use of an input fuel utilization and an input area to determine the total moles per second of hydrogen consumed and the current density.

Once the total moles of hydrogen consumed is known by one of the above three modes, the molar flow rates of both the anode and cathode flows are then adjusted to reflect the loss in moles of hydrogen at the anode and the moles of water produced and oxygen consumed at the cathode:

$$n_{a,ex,h2} = n_{a,in,h2} - n_{h2,cons}$$

$$n_{c,ex,h2o} = n_{c,in,h2o} + n_{h2,cons}$$

$$n_{c,ex,o2} = n_{c,in,o2} - \frac{n_{h2,cons}}{2}$$

where the subscript ex denotes the exit value.

The enthalpies and entropies of the anode and cathode flows at the cell's exit are then calculated from the properties code. Thus, the values analogous to the input values of $\tilde{h}_{a,in}(T_{cell})$, $\tilde{s}_{a,in}(T_{cell})$, etc., can be obtained with the subscript in replaced by ex, for exit. The Δh, Δs, and Δg for the overall cell reaction can then be determined from

$$\Delta h = h_{a,ex} + h_{c,ex} - h_{a,in} - h_{c,in}$$

$$\Delta s = s_{a,ex} + s_{c,ex} - s_{a,in} - s_{c,in}$$

$$\Delta g = \Delta h - T\Delta s$$

where all terms are at the T_{cell} temperature, and the enthalpies and entropies include the mass flow rates.

The model then makes use of one of three options for determining the actual voltage across the cell. In the table lookup option, the current density is used along with the input values of the cathode pressure to determine the actual cell voltage, v_{act}. The tables are supplied to the model as inputs. In the model option, a simple representation of the voltage-current density curves is used to determine the actual cell voltage as a function of current density, cell temperature, cathode O_2 and total pressures. This is done as follows. For current density, i, greater than 0.001 A/cm^2, and T_{cell} greater than 303.15 K,

$$v_{act} = 1.05 - 0.055\log(10^3 i) - (1.0604 - 2.493 \times 10^{-3} T_{cell})i + 0.055\log(P_{o2,in})$$

For T_{cell} less than 303.15K,

$$v_{act} = 1.05 - 0.055\log(10^3 i) - (8.966 - 2.857 \times 10^{-2})i + 0.055\log(P_{o2,in})$$

For current density less than 0.001 A/cm^2,

$$V_{act} = 1.0 + 0.055\log(P_{o2,in})$$

This actual cell voltage is then adjusted based on a limit current defined as

$$i_{lim} = 1.4 + 3.924\left(\frac{P_{o2,in}}{P_{in}} - 0.21\right) + 0.2(P_{in} - 3.0)$$

If i is less than i_{lim}, then

$$V_{act} = V_{act} + 0.1\log\left(1 - \frac{i}{i_{lim}}\right)$$

else V_{act} is taken as zero. Finally, in the voltage option, the actual cell voltage is simply taken as an input.

The voltage at a cell efficiency of 100% and at the ideal efficiency can then be determined from

$$v_{100} = -\Delta h/(2F)$$

$$v_{ideal} = -\Delta g/(2F)$$

and the ideal, voltage, and actual efficiencies from

$$\eta_{ideal} = \Delta g/\Delta h$$

$$\eta_{volt} = v_{act}/v_{ideal}$$

$$\eta_{act} = \eta_{ideal}\eta_{volt}$$

The proton exchange membrane must be kept wet for the cell to work properly; hence, the water balance within the cell is an important part of the modeling. Water is basically transported across the cell from the anode to the cathode through the passage of hydrated protons. A counter pressure gradient is often maintained across the cell from the cathode to the anode to help reduce this flow. Additionally, the anode flow can act like a drying flow and carry away some of the water. The cathode flow will also carry away the water formed through the cell reactions. To accurately model each of these mechanism would require more information than what may be available for some of the preliminary cell designs. Thus, we model all of these effects by assuming that the anode exhaust flow is just saturated with the water necessary to maintain this condition coming from the anode and cathode flows.

First the anode exhaust flow is adjusted to the saturated conditions by

$$n_{a,ex,h2o} = n_{a,tot,dry}\left(\frac{p_{vap}}{p_a - p_{vap}}\right)$$

where $n_{a,tot,dry}$ represents the total anode flow molar flow rate minus any water, p_a is the anode flow pressure, and p_{vap} is the vapor pressure of water at T_{cell}. The water that may be required, if the anode flow does not have sufficient water within it to reach this saturated condition, will be obtained from the water vapor within the cathode flow. Note that if there is insufficient water vapor within the cathode flow, then water will be added to the anode flow only up to the extent that it is available from the cathode flow. In this case, the anode flow will not be fully saturated at the exit.

Next the cathode flow is adjusted to account for the water that is either given up or required by the anode saturation condition (note that the cathode flow has already been adjusted to include the water generated within the cell reactions):

$$n_{c,ex,h2o} = n_{c,ex,h2o} + n_{a,in,h2o} - n_{a,ex,h2o}$$

The properties of the adjusted anode and cathode flow are then calculated from the properties code. An energy balance is then made over the cell from the inlet to the exit conditions. The total enthalpy change across the cell is calculated from the saved inlet enthalpies and the final enthalpies just calculated:

$$\Delta h_{tot} = \tilde{h}_{a,in} + \tilde{h}_{c,in} - h_{a,ex} - h_{c,ex}$$

where the enthalpies at the exit also include the mass flow rates. The actual electrical power generated by the cell can then be calculated from

$$P_{elec} = -\eta_{act}\Delta h$$

where Δh was calculated previously as the overall enthalpy change at the cell temperature. The heat absorbed by the coolant flow then becomes

$$Q = \Delta h_{tot} - P_{elec}$$

The model also has a bypass option, mainly for use in start-up calculations, in which all of the above calculations are bypassed with the exception of the enthalpy changes of the cathode flow across the cell and the calculations of the heat absorbed by the coolant flow. Essentially, this is equivalent to splitting the anode around the cell but still letting the cathode flow go through the cell.

Within the coolant flow entry of the model the exit conditions of the coolant flow are calculated from the above calculated value of Q. If the model is being run in the design mode, then the value of Q is simply added to the inlet coolant flow enthalpy, and the `prop` code is called to determine the properties at coolant flow exit. Additionally, a log mean temperature difference between the inlet and ext temperatures of the coolant flow and the cell temperature is calculated; an effective overall heat transfer coefficient is calculated; and based on a weight-to-power input factor, the fuel cell weight is calculated. If the cell is being run in an off-design mode, then the effective overall heat transfer coefficient as calculated in the design mode is used along with the cell surface area to obtain the exit temperature of the coolant flow. This is done by iterating the coolant flow enthalpy at the exit until the equation

$$m_{cool}(h_{cool,ex} - h_{cool,in}) = 2uA\Delta T_{mean}$$

is satisfied. Note that the same equation is used in the design mode, where the enthalpy difference for the coolant flow is obtained from the value of Q, to obtain the value of u, the effective overall heat transfer coeficient. Thus, one should not place too much significience on the value of u as it is being defined relative to the anode and cathode surface areas and not a heat transfer area.

Finally, for use in dynamic runs, the coolant flow entry will calculate the cell temperature from the following:

$$cp_{cell} W_{cell} \frac{dT_{cell}}{dt} = Q - m_{cool}(h_{cool,ex} - h_{cool,in})$$

where cp_{cell} and W_{cell} are the specific heat and weight of the fuel cell, respectively.

The parameters for the proton exchange membrane model are defined as follows:

mode -	Character string indicating either design ("d") mode, off-design ("o") mode, or ("u") utilization mode (default is "d"). Input.
option -	Two-character string giving several options for the code operation. The first character signifies the option to use in calculating the cell voltage. If specified as a "t", the table option is to be used, if "m", the model option will be used, and if "v", the voltage is input. The second character signifies whether the exit flows are to be readjusted to make the anode flow saturated if specified as "s", or not to make this adjustment if specified as anything else ("ms"). Input.
curden -	Cell current density in amperes/square centimeter (0.75). Input.
celltemp -	Cell temperature (353K). Input.
heat -	Hheat added to the coolant flow (W). Output.
fuelutil -	Fraction of input hydrogen in the anode flow consumed in the overall cell reaction (0.8). Input in the design mode; output in the off-design modes.
o2util -	Fraction of input oxygen in the cathode flow consumed in the overall cell reaction. Output.
dh -	ΔH of the cell reaction per mole of hydrogen input at the cell temperature. Output.
dg -	ΔG of the cell reaction per mole of hydrogen input at the cell temperature. Output.
voltideal -	Theoretical thermodynamically reversible cell voltage (V). Output.
volt100 -	Theoretical cell voltage based on ΔH rather than ΔG.
voltact -	Cell voltage using the voltage/current density curves (V). Output.
effideal -	ΔG over ΔH. Output.
effvolt -	Ratio of the actual voltage to the ideal voltage. Output.
effact -	Actual efficiency, η_{act}. Output.
pvap -	Vapor pressure of water at the cell temperature (atm). Output.
ph2o -	Partial pressure of water within the anode flow at the exit (atm). Output.
h2oain -	Total moles/second of water in the anode inlet. Output.
h2oaout -	Total moles/second of water in the anode exit flow. Output.
h2odiff -	Total moles/second of water crossing from the anode to cathode. Output.
hdel -	Total enthalpy change of both flows across cell (W). Output.
heatgen -	Total heat generated by the cell reactions (W). Output.
area -	Total surface area of the cell (m^2). Input in the off-design mode.
lmtd -	Log mean temperature difference between the cell temperature and the coolant inlet and exit temperatures (T). Output.
u -	Effective overall heat transfer coefficient based on lmtd, area, and heat. Output.
vtab[4][16] -	Table of voltages (V) for four cell pressure levels (atm) and sixteencurrent densities (A/cm^2) as defined by itab. Input. This is currently set to a default table of values generated by an actual fuel cell stack.

itab[4][16] -	Table of current densities (A/cm2) for the four cell pressure levels (atm) and the sixteen votages corresponding to `vtab`. Input. This is currently set to a default table of values generated by an actual fuel cell stack.
ptab[4] -	Table of pressure levels used by `vtab` (atm). Input. This is set to values required by the default `vtab`.
power -	Power structure for the model; `power.work` contains the total electrical output from the cell (W). Output.
bypass -	Flag used to bypass most of the cell calculations when set to 1 (default is 0). Input.
w_p -	Ratio of weight (kg) to power (kW) for determining the fuel cell weight (1.6). Input.
cpcell -	Specific heat of the cell for use in dynamic runs (1000 J/kg-K). Input.
weight -	Weight of the fuel cell (1kg). Input in dynamic runs.
fla -	Anode gas flow at exit.
flc -	Cathode gas flow at exit.
flcool -	Coolant water flow at exit.

5.3.22 Solid Oxide Fuel Cell (sofc) Model Class

The solid oxide fuel cell (`sofc`)model makes use of three member functions to process the anode and cathode flows to the cell. The first function, `ain`, is used to obtain the input anode flow. This entry, which should be called first, requires one input `gastype` flow and generates no output flows. The next function is `c` and is used to obtain the cathode flow and to perform the model calculations. This function requires one `gastype` flow on input and generates one `gastype` flow on output. Finally, the `a` function can be used to obtain the output of the anode side of the fuel cell. This function requires no input flows but generates one `gastype` flow and should only be called after the `c` function has been called. The `ain` and `c` functions require that a flow with the "GAS" id be used.

The modeling within the `sofc` first saves the total enthalpies (mass times specific enthalpy) of the inlet anode and cathode flows,

$$h_{a,in} = m_a h_a$$

$$h_{c,in} = m_c h_c$$

and then calls the `prop` function to evaluate the state points of the inlet flows at the cell temperature, T_{cell}. The resulting total enthalpies and entropies are then saved in $h_{a,in,Tcell}$, $s_{a,in,Tcell}$, $h_{c,in,Tcell}$, and $s_{c,in,Tcell}$. Next, the anode inlet species molar flow rates, $n_{a,in}$ are calculated from the inlet flow's `comp` array,

$$n_{a,in} = comp_{a,in} m_a$$

with a similar expression for $n_{c,in}$.

From a specified fuel utilization, U_{fuel}, the total number of moles of oxygen required for the cell reaction is calculated from

$$n_{o2} = U_{fuel}\frac{3n_{a,in,CH3OH} + 4n_{a,in,CH4} + n_{a,in,H2} + n_{a,in,CO}}{2}$$

This is compared with the available molar flow rate within the cathode flow. If there is insufficient oxygen within the flow, a message is printed, and the fuel utilization is reduced to that which the cathode flow could handle. In either case, the species molar flow rates and the overall mass flow rates for both the anode and cathode flows are readjusted to reflect the transfer of oxygen across the cell:

$$n_{a,out,O2} = n_{a,in,O2} + n_{O2}$$

$$n_{c,out,O2} = n_{c,in,O2} - n_{O2}$$

$$m_{a,out} = m_a + MW_{O2}n_{O2}$$

$$m_{c,out} = m_c - MW_{O2}n_{O2}$$

The state points of both flows are now evaluated at the cell temperature. This first requires calculating the comp array for both flows using

$$\mathrm{comp}_{a,\,out} = \frac{n_{a,\,out}}{m_{a,\,out}}$$

with a similar expression for $\mathrm{comp}_{c,\,out}$, calling the atom function, and then calling the prop function. The call to the properties code will result in the new equilibrium concentrations (i.e., comp arrays) reflecting the oxidation of the fuel components on the anode side and the loss of oxygen on the cathode side. The resulting values for the total enthalpy and entropy are then saved in $h_{a,out,Tcell}$, $s_{a,out,Tcell}$, $h_{c,out,Tcell}$, and $s_{c,out,Tcell}$. The Δh and Δg for the cell reaction can then be calculated as

$$\Delta h = h_{a,\,out,\,Tcell} - h_{a,\,in,\,Tcell} + h_{c,\,out,\,Tcell} - h_{c,\,in,\,Tcell}$$

$$\Delta g = \Delta h - T_{cell}(s_{a,\,out,\,Tcell} - s_{a,\,in,\,Tcell} + s_{c,\,out,\,Tcell} - s_{c,\,in,\,Tcell})$$

For use in calculating the Nernst potential, the partial pressures of the anode side H_2, H_2O, and cathode side O_2 are calculated from the values of $\mathrm{comp}_{a,out}$, MW_a, $\mathrm{comp}_{c,out}$, MW_c, p_a, and p_c,

$$p_{a,\,H2o} = \mathrm{comp}_{a,\,out,\,H2o} MW_a p_a$$

$$p_{a,\,H2} = \mathrm{comp}_{a,\,out,\,H2} MW_a p_a$$

$$p_{c,\,O2} = \mathrm{comp}_{c,\,out,\,O2} MW_c p_c$$

In these equations, $MW_a$and MW_c are the molecular weights of the anode and cathode flows, and p_a and p_c are the anode and cathode flow pressures. The Nernst potential is then calculated from

$$V_{nernst} = e_0 + 4.3086\times10^{-5}\left(T_a \log\left(\frac{p_{a,\,H2}}{p_{a,\,H2O}}\right) + T_c \log\left(p_{c,\,O2}^{\frac{1}{2}}\right)\right)$$

where

$$e_0 = 0.021682(57.939 - T_a(11.527\times10^{-3} + 0.6\times10^{-6} T_a))$$

and T is the temperature. The actual cell voltage is determined from

$$V_{act} = V_{nernst} - \Delta V$$

where ΔV is a specified cell overpotential and resistive loss. The cell current is calculated from

$$I = n_{o2} F$$

where F is Faraday's constant. Then,he total electrical power is calculated from

$$P_{elec} = V_{act} I$$

Several measures of cell efficiency are then calculated for use in printout. These are defined as follows

$$V_{ideal} = -\frac{\Delta g}{I}$$

$$\eta_{ideal} = \frac{\Delta g}{\Delta h}$$

$$\eta_{volt} = \frac{V_{act}}{V_{ideal}}$$

$$\eta_{act} = \eta_{ideal}\eta_{volt}$$

At this point, an overall energy balance is made by iterating the exit temperature from the cell until the total enthalpy change of both flows across the cell is equal to the total electrical power. It is assumed that both flows leave the cell at the same temperature. That is, $T_a = T_c = T_{exit}$ is varied until

$$h_{a,\,out,\,Texit} - h_{a,\,in,\,Texit} + h_{c,\,out,\,Texit} - h_{c,\,in,\,Texit} = P_{elec}$$

The parameters for the solid oxide fuel cell model are as follows:

current -	Cell current (A/m^2). Output.
celltemp -	Cell temperature at which the cell reactions are assumed to take place (1273 K). Input.
deltavolt -	Voltage drop between the Nernst potential and the cell voltage (0.02 V). Input.
fuelutil -	Specified fuel utilization (0.8). Input.
imax -	Maximum number of iterations allowed in calculating the overall cell energy balance (20). Input.
prt -	Print switch for generating debugging output (0). Input.
pf -	Pressure factor representing the fraction of inlet flow pressure to be taken as a pressure drop across the cell (0.0). Input.
fu -	Fuel utilization used by the model when there is insufficient oxygen within the cathode flow. Output.
o2util -	Oxygen utilization within the cathode flow. Output.
hdeltc -	Δh of the cell reaction at the cell temperature. Output.
gdeltc -	Δg of the cell reaction at the cell temperature. Output.
nernst -	Nernst potential (V). Output.
voltideal -	Ideal voltage of the cell (V). Output.
voltact -	Actual voltage of the cell (V). Output.
dheat -	Difference in enthalpy changes across the cell and the output electrical power (W). Output. Note that `dheat` should be zero for a correct overall energy balance.
effideal -	Ideal efficiency of the cell. Output.
effact -	Actual efficiency of the cell. Output.
power -	Power structure. `Power.work` will have the net electrical power generated by the cell (W). Output.
fla -	Anode `gastype` structure at exit. Output.
flc -	Cathode `gastype` structure at exit. Output.

5.3.23 Molten Carbonate Fuel Cell (mcfc) Model Class

The molten carbonate fuel cell (`mcfc`) model is very similar to the solid oxide fuel cell model it that it makes use of three functions to process the anode and cathode flows into and out of the model. The first is `ain`, which picks up the anode inlet flow. This function requires one `gastype` flow and generates no output flows and should be called before the others. The second function, denoted `c`, obtains the cathode inlet flow and performs the cell calculations. This function also requires one `gastype` flow on input and generates one `gastype` flow on output representing the exit cathode flow. The third function is `a`, which requires no input flows but generates one `gastype` flow representing the anode exit stream. This function should only be called after the `c` function has been called.

Since the modeling within the molten carbonate fuel cell is similar to that of the solid oxide cell, only the differences from that model will be discussed. Like the solid oxide cell, on entry to the calculational function the inlet flow enthalpies and entropies are saved, and the molar flow rates of the inlet species are calculated from the inlet `comp` arrays. The flows are then taken to the cell temperature, and the enthalpies and entropies are recalculated. The number of moles of CO_3^{--} crossing the cell due to the cell reactions is calculated from

$$n_{CO3} = U_{fuel}(3n_{a,\,in,\,CH3OH} + 4n_{a,\,in,\,CH4} + n_{a,\,in,\,H2} + n_{a,\,in,\,CO})$$

where the notation is the same as in the solid oxide model. The cathode flow is then checked to make sure that it has sufficient CO_2 and O_2 to supply the n_{CO3} moles of CO_3^{--} for the cell reactions. If it is lacking in either CO_2 or O_2, a message is printed and the fuel utilization is reduced to a level for which there is sufficient CO_2 and O_2. The species flow rates and total mass flow rates within the anode and cathode flows are then adjusted to reflect the cell reaction:

$$n_{a,out,O2} = n_{a,in,O2} + \frac{n_{CO3}}{2}$$

$$n_{a,out,CO2} = n_{a,in,CO2} + n_{CO3}$$

$$n_{c,out,O2} = n_{c,in,O2} - \frac{n_{CO3}}{2}$$

$$n_{c,out,CO2} = n_{c,in,CO2} - n_{CO3}$$

$$m_{a,out} = m_a + \left(\frac{MW_{O2}}{2} + MW_{CO2}\right)n_{CO3}$$

$$m_{c,out} = m_c - \left(\frac{MW_{O2}}{2} + MW_{CO2}\right)n_{CO3}$$

The state points for the flows are then calculated, still at the cell temperature, using calls to `atom` and `prop`. The Δh and Δg for the cell reaction are calculated as in the solid oxide model, and the partial pressures for those species involved in the Nernst potential expression are calculated based on the new molar flow rates. The Nernst potential in this case is calculated from

$$V_{nernst} = e_0 + 4.3086\times10^{-5}\left(T_a \log\left(\frac{p_{a,H2}}{p_{a,H2O}}\right) + T_c \log\left(\frac{p_{c,O2}^{\frac{1}{2}}}{p_{c,CO2}}\right)\right)$$

where

$$e_0 = 0.021682(58.2807 - T_a(0.0112811 + 9.57143\times10^{-7}T_a))$$

The rest of the cell calculations are the same as in the solid oxide cell, that is, the various measures of cell efficiency are calculated followed by an energy balance in which the exit flow temperatures are iterated.

The parameters for the molten carbonate fuel cell model are defined as follows:

current -	Cell current (A/m^2). Output.
celltemp -	Cell temperature at which the cell reactions are assumed to take place (900 K). Input.
deltavolt -	Voltage drop between the Nernst potential and the cell voltage (0.2 V). Input.
fuelutil -	Specified fuel utilization (0.75). Input.
imax -	Maximum number of iterations allowed in calculating the overall cell energy balance (20). Input.
prt -	Print switch for generating debugging output (0). Input.
fu -	Fuel utilization used by the model when there is insufficient oxygen within the cathode flow. Output.
o2util -	Oxygen utilization within the cathode flow. Output.
hdeltc -	Δh of the cell reaction at the cell temperature. Output.
gdeltc -	Δg of the cell reaction at the cell temperature. Output.
nernst -	Nernst potential (V). Output.
voltideal -	Ideal voltage of the cell (V). Output.
voltact -	Actual voltage of the cell (V). Output.

dheat -	Difference in enthalpy changes across the cell and the output electrical power (W). Output. Note that `dheat` should be zero for a correct overall energy balance.
effideal -	Ideal efficiency of the cell. Output.
effact -	Actual efficiency of the cell. Output.
power -	Power structure. `Power.work` will have the net electrical power generated by the cell (W). Output.
fla -	Anode `gastype` structure at exit. Output.
flc -	Cathode `gastype` structure at exit. Output.

5.3.24 Phosphoric Acid Fuel Cell (pafc) Model Class

The phosphoric acid fuel cell (`pafc`) model is similar to the solid oxide fuel cell model in that it makes use of three functions to handle the anode and cathode flows. The first function is `ain`, representing the inlet anode flow, and requires one input `gastype` flow and generates no output flows. The second function is `c` and represents the inlet cathode flow. This function also performs the cell calculations and generates one `gastype` flow representing the exit cathode flow. The third function is `a` and represents the exit anode flow. This function requires no input flows, generates one exit `gastype` flow, and should only be called after the `c` function has been called. In addition to these three functions, the `pafc` model also has a `cool` function, which is used to represent the coolant flow to the cell. This entry requires one input and generates one output `gastype` flow. Additionally, this function should also only be called after the `c` function has been called.

The `pafc` model is similar to that of the `sofc` model; thus only the differences from that model will be discussed. As with the `sofc` model, the inlet state points for both flows are saved, the species molar flow rates are calculated, and the flows are brought to the cell temperature, where the `prop` code is used to determine the new state points. At this point, the number of moles of H_2 consumed in the cell reaction is calculated based on an input fuel utilization,

$$n_{H2} = U_{fuel} n_{a,\,in,\,H2}$$

where the notation is the same as with the solid oxide model. The cathode flow is then checked to ascertain whether there is a sufficient oxygen flow rate for this amount of cell reaction. If the oxygen rate is insufficient, a message is printed, and the fuel utilization is reduced to a level for which there is sufficient oxygen. The species molar flow rates and mass flow rates for both the anode and cathode flows are then readjusted to reflect the cell reaction, as follows:

$$n_{a,out,H2} = n_{a,in,H2} - n_{H2}$$

$$n_{c,out,H2O} = n_{c,in,H2O} + n_{H2}$$

$$n_{c,out,O2} = n_{c,in,O2} - \frac{n_{H2}}{2}$$

$$m_{s,out} = m_a - n_{H2} MW_{H2}$$

$$m_{c,out} = m_c + n_{H2} MW_{H2O} - \frac{n_{H2}}{2} MW_{O2}$$

The `prop` code is then called to obtain the state points for the readjusted flows. The Nernst potential is calculated as

$$V_{nernst} = e_0 + 4.3086\times10^{-5}\left(T_a \log(p_{a,\,H2}) + T_c \log\left(\frac{p_{c,\,O2}^{\frac{1}{2}}}{p_{c,\,H2O}}\right)\right)$$

where

$$e_o = 0.0216816(51.147 - 0.0118984(T_a - 600))$$

The rest of the calculations are similar to those of the solid oxide model, with the exception that no iterations are performed over the exit temperature. Instead, the difference in inlet minus exit enthalpies for both flows minus the electrical power generated is treated as a heat load. This heat is then added to the coolant flow when the `cool` entry is called.

The parameters for the phosphoric acid fuel cell model are defined as follows:

current -	Cell current (A/m^2). Output.
celltemp -	Cell temperature at which the cell reactions are assumed to take place (500 K). Input.
deltavolt -	Voltage drop between the Nernst potential and the cell voltage (0.1 V). Input.
fuelutil -	Specified fuel utilization (0.8). Input.
fu -	Fuel utilization used by the model when there is insufficient oxygen within the cathode flow. Output.
o2util -	Oxygen utilization within the cathode flow. Output.
hdeltc -	Δh of the cell reaction at the cell temperature. Output.
gdeltc -	Δg of the cell reaction at the cell temperature. Output.
nernst -	Nernst potential (V). Output.
voltideal -	Ideal voltage of the cell (V). Output.
voltact -	Actual voltage of the cell (V). Output.
heat -	Difference in enthalpy changes across the cell and the output electrical power (W). Output.
effideal -	Ideal efficiency of the cell. Output.
effact -	Actual efficiency of the cell. Output.
power -	Power structure. `Power.work` will have the net electrical power generated by the cell (W). Output.
fla -	Anode `gastype` structure at exit. Output.
flc -	Cathode `gastype` structure at exit. Output.
flcool -	Coolant `gastype` structure at exit. Output.

5.3.25 Shaft (shft) Model Class

The `shft` model class is used to initiate a `shfttype` flow. Like the `gas` model, this model has `c`, `sav`, and `rec` functions for initiating a `shfttype` flow, saving a flow from the `shfts` stack, and recovering a flow (i.e., placing the flow back onto the `shfts` stack). An additional function, denoted as `end`, is used to terminate a shaft flow for dynamic simulations.

Within the `c` function, the shaft flow's inertia and power are set to zero, while the rpm is set equal to a specified input value. Within the `end` function, a new shaft rpm is calculated from the following equation

$$\Omega I \frac{\partial \Omega}{\partial t} = \left(\frac{60}{2\pi}\right)^2 Pow_{net}$$

where Ω is the shaft rpm, Pow_{net} is the net power delivered to the shaft, and I is the total moment of inertia of all the components on the shaft.

The shaft model parameters are defined as follows:

rpm -	Revolutions per minute of the shaft (1.0). Input. This parameter is recalculated within the `end` function for dynamic runs.
drpm -	Time derivative of the rpm. Output.
power -	Net power delivered by all the components on the shaft. Output.
inertia -	Net polar moment of inertia of all the components on the shaft. Output.
shftf -	Shaft flow from the model at the exit. Output.

5.3.26 Generator (gen) Model Class

The generator (`gen`) model class implements a very simple electrical generator. The model has one calculation function, `c`, that takes a `shfttype` flow from the `shfts` stack and puts one `shfttype` flow back onto the stack.

The model simply defines the power consumed from the shaft (and hence, generated as electrical output) from

$$\mathrm{Pow} = \mathrm{Pow}_{\mathrm{rated}}\left(\frac{\mathrm{rpm}}{\mathrm{rpm}_{\mathrm{rated}}}\right)^4$$

where $\mathrm{Pow}_{\mathrm{rated}}$ and $\mathrm{rpm}_{\mathrm{rated}}$ represent some input rated power and rpm, and `rpm` is the input shaft rpm. This power is then removed from the shaft and an additional moment of inertial is added to the shaft. As an option, the generator model can be made to consume all of the power available within the shaft flow.

The generator model parameters are defined as follows:

inertia -	Moment of inertia for the generator (0.2). Input.
rat_rpm -	Rated rpm of the generator (3600). Input.
rat_pow -	Rated power of the generator (40 kW). Input.
stat -	Flag, if set to 1, will tell the generator to consume all of the shaft power (0). Input.

5.3.27 Motor (mot) Model Class

The motor (`mot`) model class is used to simulate a very simple electrical motor. The model has one calculational function, `c`, that takes one `shfttype` flow from the `shfts` stack and puts one `shfttype` flow back onto the stack.

Several options are provided for defining the power added to the shaft flow. The first option, denoted as "zero", simply adds to the shaft power a value equal to the negative of the power currently within the shaft. This option simply adds (or subtracts) the power necessary to prevent the rpm of the shaft from changing. The second option, denoted as "level", will add additional power to the shaft to bring it up to a specified power level, P_0, if the input shaft power is greater than zero but less than the specified power level. If the shaft already has a power of P_0, then no additional power is added. If the shaft has a negative power level, then the model adds to the shaft a power of P_0. Finally, a third option simply adds a specified power to the shaft flow. The model also adds to the shaft flow a specified moment of inertia.

The motor model parameters are defined as follows:

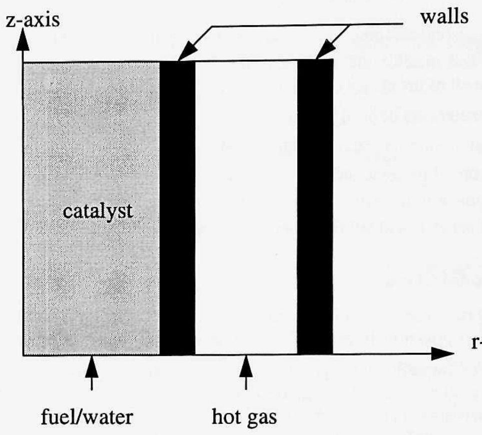
inertia -	Moment of inertia for the motor (0.1). Input.
power -	Input power (0.0). Input when option is "".
power0 -	Specified power level when option is "level" (10^4). Input.
option -	Character string defining the option (""). Input.

5.3.28 Detailed Steam Reformer (refs) Model Class

The dynamic `refs` model class simulates a methanol steam reformer. The model is based on a detailed stand-alone model that has been simplified slightly for use in the system simulations. The model calculates the temperatures and species profiles, not only along the flow path, but also normal to the flow path. Thus, `refs` is a two-dimensional, time-dependent model. The model has several functions for processing the flows. The first is `hin`, representing the input flow on the hot side, and takes one `gastype` flow from the `gass` stack. The second function is `cin`, representing the fuel/water input flow, which also takes one `gastype` flow. Neither `hin` nor `cin` generates any output flows. Once both `cin` and `hin` have been called, the model will automatically call another function, denoted as wk. This function will perform the model calculations but, will not generate any output flows. The output flows for the hot and cold sides are then obtained by calling `hout` and `cout`, respectively. Both of these functions generate one `gastype` flow. At present only one instance of the `refs` class can be used in any simulation due to the large number of variables that the coding uses. Since this model is more detailed than the others we present the modeling in a number of subsections.

Reformer geometry

The basic geometry used with the reformer model is either cylindrical or cartesian. In the cylindrical geometry the coordinates are (r, θ, z) with the flow passages parallel to the z axis. Thus, both the reformate gas (initially the inlet fuel/water mixture) and the hot gas passages are either circular or annular in cross section. At present, only one reformate flow passage is permitted through the reformer, and its direction defines the direction of the z-coordinate. Rotational symmetry about the z-coordinate axis is assumed. In the radial or r-coordinate direction, a number of different regions are present. These include the catalyst bed (which also represents the reformate gas region), the hot gas, walls, and/or void regions. The following figure shows a typical situation with the catalyst/reformate gas within a central cylindrical tube surrounded by an annular hot gas flow. Note that the model can also handle the hot gas inside an annular



Typical reformer geometry

tube of catalyst or both inside and outside the catalyst. Generally, the reformer would be made up of a number of such catalyst tubes. It is assumed that all such tubes are the same.

In the cartesian geometry, a flat plate reformer is modeled. Here the coordinates are (x,y,z), with the z axis defining the flow direction. The x direction has three regions defining the reformate, wall, and hot gas flows. The y direction represents the width of the reformer flow passages. The reformate and hot-gas flow regions are taken only to their center lines. As with the cylindrical geometry, a number of pairs of reformate and hot gas flow paths make up the reformer, with all such flow pairs assumed to be the same.

Gas properties

Due to the large number of calculations made within the `refs` model, the thermodynamic properties calculations are performed internally, rather than by use of `prop`. The reformate flow is treated as an ideal gas with varying compositions. The species considered are CH_3OH, CO_2, CO, H_2, H_2O, O_2, and N_2. The last two are included for handling the hot gas flow and would not, usually, be found within the fuel/water reformate stream. For the hot gas flow, frozen chemistry is assumed. The thermodynamic properties of the reformate flow are calculated on the bases of the properties of the individual species. Each species, i, has an enthalpy, h_i (J/kg), defined from

$$h_i = cp_i T + h_{0i}$$

where cp_i is the specific heat (J/kg-K), T is the temperature (K), and h_{0i} is a constant, such that at some reference temperature, h_i becomes equal to the enthalpy of formation of the species at that temperature. The specific heats and enthalpies for the gas stream are then calculated from

$$cp = \sum y_i cp_i$$

$$h = cp\ T + h_0$$

where

$$h_0 = \sum y_i h_{0i}$$

and y_i represents the species mass fractions. The reference temperature chosen for defining cp_i and h_{0i} is taken as some intermediate temperature value within the gas streams. For the hot gas flow, the properties are calculated in a similar way when the flow has the "GAS" property id. Otherwise, the `prop` function is called to get the value of `cp` and density for the flow, which are then kept fixed along the flow path.

The thermal conductivity, κ, of a flow stream is calculated from the following.

$$\kappa = \sum_i \frac{x_i \kappa_i}{\sum_j x_1 \Phi_{ij}}$$

where x_i is the species mole fraction, κ_i is the species thermal conductivity, and Φ_{ij} are interaction coefficients defined by

$$\Phi_{ij} = \frac{\left(1 + \left(\frac{M_j}{M_i}\right)^{\frac{1}{4}} \sqrt{\frac{\mu_i}{\mu_j}}\right)^2}{\sqrt{8\left(1 + \frac{M_i}{M_j}\right)}}$$

Here M_i is the molecular weight, and μ_i is the viscosity of species i. As with the specific heats, the species thermal conductivity and viscosity are taken as constants at the reference temperature.

The density, ρ, is defined using the ideal gas equation of state

$$P = \frac{\rho R T}{M}$$

where R is the ideal gas constant and M is the molecular weight of the gas steam given by

$$M = \sum x_i M_i$$

For the flows on the hot gas side not having the "GAS" property id, the code makes use of input values of the flow's conductivity, with the density being calculated from `prop` as indicated above.

Conservation of mass equation

Since the main thrust of the model is to examine the thermal response of the reformer, some simplifying assumptions have been made concerning the flow fields. Firstly, it is assumed that the flow velocity is only in the $\pm z$ direction, with no other component of velocity being calculated. Secondly, it is assumed that the pressure drops within the flow fields are small relative to the inlet pressures. Thus, the pressures are taken as constant throughout the flow fields. With these two assumptions, the velocity fields can be calculated using the overall conservation of mass equation alone without the use of the conservation of momentum equation. It should be noted that these assumptions do no preclude using the model to examine dynamic changes of the inlet flow conditions.The model will still correctly follow changes within the inlet fuel/water compositions, which are propagated through the reformer essentially with the velocity of the flow. Changes within the inlet velocity field, however, will tend to propagate through the reformer with approximately the sonic velocity. Thus, only for extremely short time scales (roughly on the order of L/s, where L is the reformer length, and s is the sonic velocity) would a treatment using the full conservation of momentum be needed.

The overall conservation of mass equation for a time-dependent, three-dimensional flow field is given by

$$\frac{\partial \rho}{\partial t} + \nabla \bullet (\rho \bar{v}) = 0$$

With the above assumption of a one-dimensional flow field, this equation is used to calculate, not the density, but the axial component of velocity, $\bar{v}$. The density is determined from the equation of state using the pressure, temperature (determined from the energy equation), and composition (determined from the species conservation equations).

Species and atoms conservation equations

Some additional assumptions have been made with regards to the species conservation equations. Firstly, the full multi-species convective-diffusion equations are not used. For the hot gas flow it has already been mentioned that frozen chemistry is used. This stream being of lesser importance in that it simply supplies the heat for the process, and thus, its speciation is not all that important. For the reformate flow the convective terms are far more important than the diffusion terms. Thus, a convective-diffusion equation is used with a single-species diffusion coefficient. It is furthermore assumed that the species mass flux can be written in terms of the gradient of species concentration. With these assumptions, the species conservation equation takes the form

$$\varepsilon\frac{\partial C_i}{\partial t} + \varepsilon\nabla\bullet(\bar{v}C_i) = \nabla\bullet(D\nabla C_i) + r_i\rho_{bed}(1-\varepsilon)$$

where C_i is the i'th species concentration (mol/m^3), ε is the porosity of the catalyst bed, ρ_{bed} is the density of the catalyst bed material,[†] D is the species diffusion coefficient (m^2/s), and r_i is the source term representing the per kilogram of catalyst times the rate of species creation (mol/kg-s). When integrated over a control volume, the terms on the left-hand side of this equation represent the explicit time rate of change of species i and the net convection of species into and out of the control volume. The terms on the right-hand side then represent the diffusion of species through the control volume and the source of species i due to reactions. The r_i terms will be discussed below. Within the computer model, due to the much greater effect of convection over that of diffusion, the component of diffusion in the direction of the flow has been ignored.

This species conservation equation is used for the methanol concentration and optionally, for the water concentration, depending on whether or not the water/gas shift reaction is included and taken to be in equilibrium. Note that some studies tend to indicate that the water/gas shift reaction does not occur over the copper and zinc oxide catalyst used when methanol is present. This option is included only for studying the importance of the water/gas shift on the conversion rates and final hydrogen concentrations when the catalyst reaction mechanism includes the water/gas shift.

To determine the species concentrations that are in chemical equilibrium, an equation of atom conservation is used. Defining a total atom concentration (atoms/m^3) from

$$B_i = \sum a_{ij}C_j$$

where a_{ij} is the species stoichiometric coefficient, one obtains

$$\varepsilon\frac{\partial B_i}{\partial t} + \varepsilon\nabla\bullet(\bar{v}B_i) = \nabla\bullet(D\nabla B_i)$$

Solving this equation for the B_i's, the equilibrium concentrations can be obtained. For the five species used within the reformate gas (CH_3OH, CO_2, CO, H_2, and H_2O), two cases are considered depending on whether the water/gas shift reaction is treated kinetically or in equilibrium. If treated kinetically, then only the CO_2, CO, and H_2 species need to be determined from B_i. Since there are three B_i's, one for each of carbon, hydrogen, and oxygen, these three concentrations are directly determined from the three B_i's given the values for C_{CH3OH} and C_{H2O} determined from the species conservation equations. If the water/gas shift reaction is in equilibrium, then the species conservation equation is not used to determine C_{H2O}. However, in this case, one has the equilibrium condition

$$K_{eq} = \frac{C_{CO}C_{H2O}}{C_{CO2}C_{H2}}$$

where K_{eq} is the equilibrium constant, which depends on the local temperature. This equation along with the values of B_i can then be used to determine the concentrations of CO_2, CO, H_2, and H_2O.

† Note that the porosity here is defined in terms of the void space through which the fluid flow may pass. Thus, the bed or catalyst material density would include any micro-porosity.

Conservation of energy equation

The formulation of the energy conservation within `refs` makes an approximation based on the very high effective heat transfer between the reformate gas and the catalyst bed. The temperature difference between the gas and the catalyst tends to be very small, and only within a short distance of the fuel/water inlet is there usually any substantial difference. Thus, a single temperature is used to represent both the reformate gas and that of the catalyst bed. This can produce a reasonably accurate solution with much less computational time. Additionally, the single-temperature option eliminates the need to determine both the effective heat transfer between the gas and catalyst and the relative amounts of heat that would be taken from the gas and catalyst in supplying the heats of reaction.

Due to the assumption of constant pressure, the time-dependent conservation of energy equation can be formulated as a conservation of enthalpy equation. When written in terms of the common gas and bed temperature, T, this equation becomes

$$\frac{\partial}{\partial t}[(1-\varepsilon)\rho_{bed}cp_{bed}T + \varepsilon\rho_{ref}(cp_{ref}T + h_0)] + \nabla\bullet(\bar{v}\varepsilon\rho_{ref}(cp_{ref}T + h_0)) = \nabla\bullet(\kappa\nabla T)$$

where ε is the catalyst bed porosity as before, κ is the effective bed thermal conductivity (W/m-K) , cp_{ref} is the specific heat of the reformate, cp_{bed} is the specific heat of the bed, and ρ_{bed} is the density of the bed material (kg/m^3). In this equation the terms on the left-hand side represent the explicit time-dependent changes to the enthalpy and the convection of enthalpy within the bed. The term on the right-hand side represents the heat transfer within the bed. Note that due to the way the enthalpy of the flow stream is calculated, with the species referenced to their heats of formation, heat of reaction terms do not explicitly appear within this equation.

For the hot gas flow, the energy equation is

$$\frac{\partial}{\partial t}[\rho(cp\ T_{gas} + h_0)] + \nabla\bullet[\rho\bar{v}(cp\ T_{gas} + h_0)] = \nabla\bullet(\kappa_{gas}\nabla T_{gas})$$

where cp and ρ refer to the specific heat and density of the hot gas flow, respectively.

For the walls surrounding the reformate and the hot gases the energy equation is

$$\rho_{wall}cp_{wall}\frac{\partial}{\partial t}(T_{wall}) = \nabla\bullet(\kappa_{wall}\nabla T_{wall})$$

where cp_{wall} is the specific heat, κ_{wall} is the thermal conductivity of the wall, and ρ_{wall} is the density of the wall material.

Note that the thermal conductivity used in the above equations for the flow on the hot gas side is actually an effective conductivity enhanced by a factor to account for the fluid motion. This is done, presently, by calculating an effective Nusselt number for the flow and enhancing the thermal conductivity by that number. The Reynolds number is based on the flow passage width. The skin friction and Nusselt number are calculated from the following correlations:

$$f = 2\left(\left(\frac{8}{Re}\right)^{12} + \frac{1}{(a+b)^{1.5}}\right)^{\frac{1}{12}}$$

where

$$a = \left[4.920 - 2.457\log\left(\frac{42.683}{Re^{0.9}}\right)\right]^{16}$$

$$b = \left(\frac{37530}{Re}\right)^{16}$$

and

$$Nu = b_1^{0.1}$$

where

$$b_1 = 3.657^{10} + \left(\frac{e^{\frac{2200-Re}{365}}}{3.657^2} + \frac{1}{a_1^2}\right)^{-5}$$

$$a_1 = 4.8 + \frac{0.079\sqrt{\frac{f}{2}}RePr}{(1+Pr^{0.8})^{\frac{5}{6}}}$$

For the plate geometry or for flows not having the "GAS" property id, the code permits a direct input of the Nusselt number.

Reaction kinetics

The reaction rates r_i appearing within the species conservation equations are taken from the experimental work of Amphlett et al. [7-10] and Jiang et al. [11]. In the work of Amphlett, two reactions are assumed to form the primary mechanisms of methanol conversion. These are the methanol decomposition reaction

$$CH_3OH \leftrightarrow CO + 2H_2$$

and the water/gas shift reaction

$$CO + H_2O \leftrightarrow CO_2 + H_2$$

For the methanol decomposition reaction using the G66B catalyst, the reaction rate r_{CH3OH}, is defined by the equation

$$-r_{CH3OH} = \frac{k_{ch3oh}(C_{CH3OH} - C_{CO}C_{H2}^2/K_{eq})}{1 + k_{CO}C_{CO}}$$

Here k_{ch3oh} and k_{CO} are correlated from experimental data as

$$k_{ch3oh} = 7.036x10^6 e^{-\frac{11568}{T}}$$

$$k_{CO} = 10^4 e^{-\frac{3038}{T}}$$

and the equilibrium constant is given as

$$K_{eq} = 4.275x10^{14} e^{-\frac{11160}{T}}$$

These values are in units such that given the compositions in g-mol/m^3, the reaction rate r_{CH3OH} is in g-mol/kg-s. Note that the reaction rate is per kilogram of catalyst material. The experimental work was done over a temperature range of 423 to 523 K.

For the C18HC catalyst, the reaction rate is defined by the following equation:

$$-r_{CH3OH} = k_{ch3oh}(C_{CH3OH} - C_{CO}C_{H2}^2/K_{eq})$$

where

$$k_{ch3oh} = 8.06x10^5 e^{-\frac{10100}{T}}$$

The experimental data used to determine this correlation were obtained over a temperature range of 423 to 543 K.

For the water/gas shift reaction over the G66B catalyst, Amphlett et al. indicated some uncertainty in the data above 453 K. Because of these uncertainties, the reaction rates are only tentative, until additional information is obtained. The water/gas shift reaction rate ($-r_{H2O}$) is thus defined by the following equation:

$$-r_{H2O} = k_{h2o}(C_{H2O}C_{CO} - C_{CO2}C_{H2}/K_{eq})$$

where

$$k_{H2O} = 6.83762 \times 10^{6} e^{-\frac{11567}{T}}$$

and the equilibrium constant for this reaction is

$$K_{eq} = 8.06 \times 10^{-3} e^{\frac{4880}{T}}$$

As with the methanol decomposition, the units are such that with compositions in g-mol/m^3 the reaction rate is in g-mol/kg-s. No information was given concerning the reaction rates of the water/gas shift using the C18HC catalyst; thus, at present, the same expressions for the G66B catalyst are being used within the model.

For use with another catalyst (BASF S3-85), Jiang et al.[11] proposed a different set of reaction mechanisms for the reforming of methanol. In particular, their studies tend to suggest that the water/gas shift reaction does not occur in the presence of methanol. In this work the overall methanol reformation was determined to follow the reaction rate equation:

$$-r_{CH3OH} = 5.31 \times 10^{9} e^{\frac{-1.05 \times 10^{5}}{RT}} P_{CH3OH}^{0.26} P_{H2O}^{0.03} P_{H2}^{-0.2}$$

where P_i is the partial pressure in kPa. This expression was obtained experimentally over a temperature range of 443 to 533 K and at atmospheric pressure. The same rate expression is also used for $-r_{H2O}$.

Note that all experimental work tended to show that, at least, for sufficiently small catalyst particles the reaction rates were independent of the particle size. In the case of Amphlett's work the reaction rates were for the commercial pellet size down to 20-24 mesh. For Jiang's work the particle size variation was between 150 and 590 μm.

Boundary and initial conditions

The boundary conditions used for the above conservation equations consist of symmetry conditions along the cylinder axis or the center line of the flow for plate geometries and a fully insulated condition along any outer-containing wall. Thus, along r=0 (or x=0) and $r=r_{max}$ (or $x=x_{max}$), we have $\frac{\partial T}{\partial r} = 0$. We also assume that the wall regions along the cylinder ends are fully insulated, thus, $\frac{\partial T}{\partial z} = 0$ over these faces. For the fluid flow regions, the temperature, as well as pressure and compositions, are all known along the inlet. For the outlet, a high Pecklet number is assumed, essentially yielding $\frac{\partial \phi}{\partial z} = 0$ for $\phi = T, C_i, B_i$,and u. Note that no heat flux boundary conditions are imposed between the separate regions, such as between the walls and the fluid flows, since the temperature field is calculated as a single field over the entire reformer. The numerical technique employed, however, permits jumps within the effective thermal conductivity of the materials, which essentially forms a condition that the heat flux between regions is continuous. For the species concentrations and the total atom concentrations, no diffusion occurs at the bounding walls, thus, for these radial values, $\frac{\partial C_i}{\partial r} = 0$ and $\frac{\partial B_i}{\partial r} = 0$ and similiar conditions for the plate geometry. For the constant temperature mode of operation no boundary conditions are, of course, needed for the temperature fields, but the boundary conditions on C_i and B_i are the same as described above.

For dynamic problems, the initial conditions are obtained by solving the above equations with the time derivatives set to zero.

Refs model parameters

length -	Length of a flow passage along the reformer (1 m). Input.
width -	Width of a flow passage when `geom` is "p" or 2π when geom is "c" (1 m). Input in when `geom` is "p".
ntubes -	Number of reformer tubes or number of pairs of reformate and hot gas flows (200). Input.
kbed -	Effective thermal conductivity of the bed (0.7 W/m-K). Input.
kwall -	Thermal conductivity of the wall material (20 W/m-K). Input.
porbed -	Porosity of the bed (0.3). Input.
cpbed -	Specific heat of the bed material (1100 J/kg-K). Input.
cpwall -	Specific heat of the wall material (500 J/kg-K). Input.
rbed -	Density of the bed material (1930 kg/m^3). Input.
rwall -	Density of the wall material (8000 kg/m^3). Input.
diff -	Species diffusion coefficient (10^4 m^2/s). Input.
w_f -	Catalyst weight to fuel flow rate (500 kg-s/g-mol). Input, but only used to determine a reformate flow rate when the input mass flow rate is not set within the inlet flow.
mg_mr -	Hot gas flow rate to reformate flow rate (3.2). Input, but only used to determine a hot gas flow rate when the input mass flow rate is not set within the inlet flow.
hot_cond -	Input value of the thermal conductivity of hot gas (20 W/m-K). Input, but only used when the hot gas flow does not have the "GAS" property id.
hot_nu -	Input value of the Nusselt number for hot gas flow (1.0). Input, but only used when the hot gas flow does not have the "GAS" property id.
geom -	Character string of either "p" for plate type geometry or "c" for cylindrical geometry ("c"). Input.
kinetics -	Character string defining the various kinetic equation options within the model. The first character defines the kinetics of the methanol decomposition reaction and can be either 'G' for the G66 catalyst, 'C' for the C18H8 catalyst, 'B' for the BASF catalyst, or 'I' for an input reaction rate constant. The second character represents the reaction kinetics for the water/gas shift reaction and can be either 'S' for using the same rate as for the methanol decomposition reaction, 'A' for the reaction rate from Amphlett's work, ' I' for using a specified input reaction rate constant, or 'E' to use the water/gas shift at equilibrium ("CA"). Input.
kch3oh -	Rate constant for the methanol decomposition reaction (0). Input when kinetics[0]='I'.
kco -	Constant used in the methanol decomposition reaction (0). Input when `kinetics[0]='I'`.
kh2o -	Rate constant for the water/gas shift reaction (0). Input when `kinetics[1]='I'`.
region[6] -	Region structure defining the region types, thicknesses, and number of nodes for up to six regions. Only the region parameters of `type`, `n`, and `thick` need to be input. The parameter `type` takes one of the values "R" for reformate, "W" for wall, "G" for hot gas, "V" for void space, "S" for symmetry line; `n` defines the number of nodes to be used within the region. The total number of nodes for all regions must be less than 20. The parameter `thick` defines the thickness of the region in meters. Input.
noregions -	Total number of regions defined by the region array (4). Input.
relax_t,relax_s -	Relaxation factors used in solving the temperature and species equations respectively (0.3 and 0.3). Input.
acc_t -	Accuracy criteria for terminating the iterations on temperature (10^{-3}). Input. The iterations will terminate when the maximum temperature difference between iterations is less than this parameter.

acc_s -	Accuracy criteria for terminating the iterations on the species concentrations (10^{-10}). Input.

Note that the basic geometry is dictated by the geom, noregions, and the region array parameters. These default to a cylindrical geometry with four regions consisting of a "R", "W", "G", and another "W", each with only a single node and thicknesses of 0.01, 0.001, 0.01, 0.0001 m, respectively. The outer most region is automatically defined as an "S" or symmetry line. This basically says, that no heat is transferred across the line. Note that if a plate type reformer is desired, then one should define only three regions of "R","W", and "G", where the "R" and "G" region thicknesses define the space from the wall to the center line of the region.

5.3.29 Power Stack (pows) Model Class

The power stack (pows) model class is somewhat different than the other models in that it does not process a particular flow, but makes use of a power class used by the other models. Each model that produces work, such as the gas turbine, or consumes work, such as the pump, will record this information in the work parameter of a power class. Similarly, models that input heat, such as the combustor, or lose heat from the system, such as a ht model with a negative heat load, will record this information in the heat parameter of this power class. As with the other stack classes, modstack and gasstack, a special instance of the powstack class is used by the models to store these power structures. This instance must be named pows and, like the mods and gass stacks, is pre-defined within the header file so the user does not need to explicitly define it. Each model's power structure is then put onto the pows by calling the put member function of the stack class with the model's power class as an argument. The pows model then makes use of this stack to calculate the net work and heat associated with the entire system by calling its c member function.

The model has no input parameters, but does calculate the following output parameters.

prod -	Sum of all the positive power.work variables of all models.
cons -	Sum of the absolute values of all negative power.work variables of all models.
input -	Sum of all positive power.heat variables of all models.
loss -	Sum of the absolute values of all negative power.loss variables of all models.
netprod -	Net power produced equal to prod-cons.
netinput -	Net energy input equal to input-loss.

Note even if the pows.c function is not called, the pows stack is available for printing out tables of the models' power classes via pows.print.

6.0 Steady-State Examples

In this section, six examples are given showing typical usages of the steady-state models defined in Section 5. The first three of these are very simple examples designed to show the use of flows, system constraints, and parameter studies. The next two are slightly more realistic but still relatively simple examples: a space power system and a conventional coal-fired steam plant. Finally, the last example is of a proton exchange membrane fuel cell system with the fuel supplied using a methanol steam reformer.

6.1 Gas Turbine System

Consider a system consisting of a hydrogen tank, a compressor, a heater, and a gas turbine connected together in that order. Such a system could be analyzed using the following input to GCtool.

```
modstack mods;
gas   gas1={id="THR-tH2"; t=300; p=1; m=1;};
cp    cp1={pres=6.0; eff=0.88;};
ht    ht1={temp=1000;};
gt    gt1={pres=1.0; eff=0.84;};

gas1.c;  cp1.c;  ht1.c;  gt1.c;
gass.print; mods.print;
```

First an instance of the modstack class, mods, is defined. This model class instance will be used to call all of the other model print functions. To represent the hydrogen tank we use class gas (gas flow initiator) and define a specific instance of that class as gas1. Parameter values are then assigned to initialize the flow. These include defining the flow as a hydrogen gas using "THR-tH2"[†], and then defining the values of temperature, pressure, and mass flow rate using t, p, and m, respectively. Instances of the compressor, heater, and gas turbine classes are then defined with their associated parameter values. Note that all input parameters have default values and, thus, could be left out if the defaults are appropriate for the problem. Finally, the calculational functions for each model instance are called in the order necessary for representing the system, and the gass stack and mods stack print functions are called to obtain the results. As can be seen, the largest part of the coding is supplying the model parameter values, which is often the case.

Before defining more complicated system configurations, we need to consider the mechanism for handling the fluid flow (or for that matter any type of flow) between the models. Basically, each model will take off of a flow stack (for gastype flows this is the gass stack) the number of input flows that it requires and will put back onto the stack the output flows that it generates. Actually, only the addresses of the flows are saved on this stack, but the concept is the same. Thus, in this example, the gas1.c model, being an initiator of a flow, simply puts one output flow onto the stack; cp1.c then takes this flow off the stack and, on completion of its calculations, puts its output flow back onto the stack. Models ht1.c and gt1.c then do exactly as the cp1.c model, taking their single input flow off the stack and putting their single output flow back onto the stack.

In order to handle arbitrary system configurations, where some flows may not be used by the next model in the flow path, one must be able to remove a flow from the stack, store it, and, at a later point, place that flow back on the stack for further processing. This saving of a flow is usually done by the model that will require the flow at a later point by use of an additional entry point to the model. For example, a flow mixer model, mx, requires two flows for its calculations. The first input flow is pulled from the stack and stored, using the model's secondary entry, mx.s. At a later point within the calculations, the model's calculational entry, mx.c, would then be called. This entry will obtain from the stack the second input flow and then calculate, using the first input flow which had previously been retrieved, the resulting output flow. This output flow is then placed onto the stack for further downstream components. Note that the model specified directly after the mx.s entry should be one that does not require an input flow, as the mx.s entry does not generate one.

As an example where a model generates two output flows, the gas flow splitter model, sp, makes use of two entries. The calculation entry sp.c would obtain from the gass stack the one input flow that the model needs to perform

[†]The "THR-tH2" represents a hydrogen gas flow; the small 't' before the H2 denotes a special version of the hydrogen data valid for high temperatures.

its flow splitting. Internally, the splitter model then calculates both flows but only puts one of these onto the gass stack. At a later point within the system configuration, the splitter's secondary entry, sp.s, will put onto the gass stack the second or split-off flow for use within the next model.

In addition to the gastype flow class, other types of flows may exist. For example, some of the dynamic models make use of the shfttype flow class, representing the power extracted or delivered to a model by a shaft. The different flow stacks are entirely independent of each other. Thus, in determining the input flows to a model, consideration must be given to the previous model generating an output flow of the correct flow type. For example, a shaft flow might be generated, and then many models might be called requiring only gastype flows before the model that requires the shfttype flow is called.

In general this stack mechanism means that at any point where a model is called, one needs to look at the previous models to determine what the input flows are. There are no hard and fast rules here; thus, the user of a model must have an understanding of what flows a model needs and what flows it generates. For example, one might have a collection of models that requires two input flows of the same type, both of which are placed on the stack as outputs for use in the next model.

Flows that are generated by a model and not used immediately by a subsequent model before other flows (of the same type) are generated are effectively lost to further models downstream. Thus, new models that are developed and added to GCtool should be considered from the user's point of view with regard to flow usage in order that they are not too difficult to use. Ultimately, however, it is the responsibility of the user to correctly sequence the models to represent the system configuration. Note that within the GCtool environment, a rehash and draw can be done periodically while the system inputs are being developed to "see" the system that is being constructed.

6.2 Gas Turbine System with Fixed Net Power Constraint

The first example shows how to set up a very simple gas turbine system. Here we extend that system to include a constraint. As an example we constrain the net power generated by the system to be fixed at some value, say 40 MW. Constraints, such as this one, which depend on more than one component within the system, will be called "system constraints", as opposed to "model constraints", which can be handled internally within a model. To keep the code general, system constraints are not automatically established by some built-in procedure. Additionally, constraints may often be established in more than one way. For example, in this case, one might be able to establish this power constraint by varying the pressure levels or by varying the mass flow rate. In any case, imposing the constraint is not difficult and amounts to adding an equation-solving task (see Section 4.2), where the constraint equation is the system power equal to 40 MW and the parameter to be varied must be defined by the user. In this case, we will take the parameter to be varied as the mass flow rate - gass1.m. Adding this task to the input of the first example, gives the following:

```
modstack mods;
gas    gas1={id="THR-tH2"; t=300; p=1; m=1;};
cp     cp1={pres=6.0; eff=0.88;};
ht     ht1={temp=1000;};
gt     gt1={pres=1.0; eff=0.84;};
task   a;

while (a.c)
    {vary(gas1.m, 1, 0.1, 100.);
     gas1.c;  cp1.c;  ht1.c;  gt1.c;
     cons(gas1.m, cp1.power.work+gt1.power.work-40e6);
    }
gass.print; mods.print;
```

Here, the task added is denoted as a, and gas1.m (mass flow rate) is varied between 0.1 and 50.0 with an initial value of 1. After the model calculational entries are called, the value of the power consumed by the compressor, which is denoted as cp1.power.work, and the value of the power generated by the gas turbine, gt1.power.work, will be known and the constraint can then be specified. The net power produced is the sum of these power.work substructures (remember that power.work is an algebraic quantity and hence will be negative for the compressor). When the task has converged, the constraint will be equal to zero (to the default error tolerance, since none was specified).

6.3 Gas Turbine System with Parameter Study

Continuing with the preceding example, one might desire a particular exit turbine temperature or some other constraint. These problems are solved by simply using additional vary and cons operators. Or one might want to optimize the system efficiency subject to various equality and inequality constraints. These problems also are solved by simply using additional vary, cons, icons, and mini operators as in the examples in Section 4.2. Here we add a parameter sweep to the previous problem. Suppose it is desired to look at the previous system for different heater exit temperatures of, say, 800, 1000, 1200, and 1400 K. This is accomplished by simply putting a for loop around the task loop and the gass and mods print functions. The input in these case would be as follows:

```
modstack mods;
gas    gas1={id="THR-tH2"; t=300; p=1; m=1;};
cp     cp1={pres=6.0; eff=0.88;};
ht     ht1={temp=1000;};
gt     gt1={pres=1.0; eff=0.84;};
task   a;
for (ht1.temp=800; ht1.temp<=1400; ht1.temp+=200)
    {while (a.c)
         {vary(gas1.m, 1, 0.1, 100.);
          gas1.c;  cp1.c;  ht1.c;  gt1.c;
          cons(gas1.m, cp1.power.work+gt1.power.work-40e6);
         }
     gass.print; mods.print;
    }
```

6.4 Space Propulsive System

Next we consider a somewhat more realistic example, a simple space propulsive system, a diagram of which is shown in Fig. 1. System diagrams such as shown in the figure can be semi-automatically generated through the use of GCtool. First, we consider the system formulated without any constraints.

In this example, a gastype flow is initialized by using an instance of the gas model, which has been named gas_h2. As a convention in naming the models, we will use the model class type, followed by an underscore and a label. Although the flow being initialized is a gastype, as explained previously, this really refers to the structure of the flow class type, not that the flow needs to be in the gas phase. In this case, the gas_h2 model parameters will be defined to initialize a hydrogen flow within the liquid region. This hydrogen flow is then passed through low- and high-pressure pumps, denoted pump_lp and pump_hp. The flow then passes through a heat exchanger hx_nz, representing the nozzle cooling. Note, the current nozzle model does not include the provisions for a coolant flow; thus, this heat exchanger is used to simulate these effects.

The flow is then split using sp_2 into a main flow and a secondary flow, which is further split using sp_1. These last two flows are then passed through two gas turbines, gt_hp and gt_lp, which are used to drive the low- and high-pressure pumps. These gas turbine flows are then mixed together in mx_1 and then mixed back into the main flow in mx_2. The resulting flow is then passed through a heater model used to simulate a reactor, denoted as ht_reac and then through the hot side of the hx_nz model and out the main thruster nozzle, nz_1.

In formulating the inputs we will start with the model calls necessary to describe the system configuration. This will be done exactly like the simpler examples described above by simply listing the models in the order that they process the gastype flows and using the secondary splitter and mixer functions where necessary. Note, depending on which flows from the splitters are treated as the primary flow and which are treated as the split-off or secondary flow, different system representations can be defined. Here we will assume that the primary flow from sp_2 passes through sp_1, and that the primary flow from sp_1 passes through the gt_lp model. Thus, the system is described up to the secondary function of the mx_1 model by

```
gas_h2.c; pump_lp.c; pump_hp.c; hx_nz.c; sp_2.c; sp_1.c; gt_lp.c; mx_1.s;
```

At this point, the secondary function of the sp_2 model can be called to retrieve its split-off flow, which is then processed into the secondary function of the mx_2 model using

```
sp_2.s; mx_2.s;
```

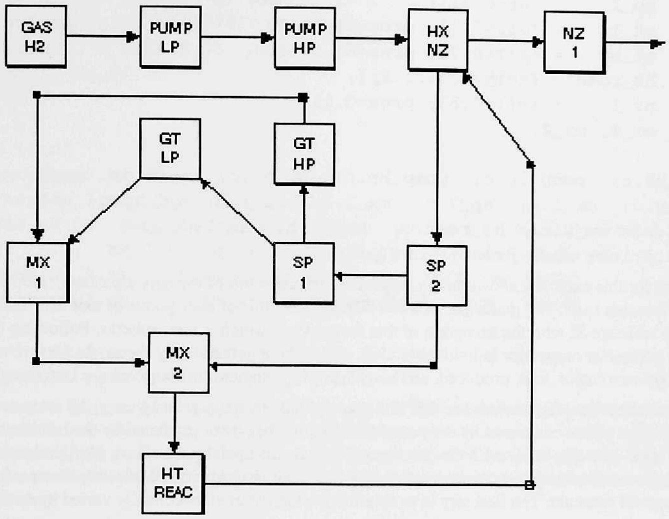


Fig. 1. Space Propulsive System

Then the secondary function of the sp_1 model can be called to retrieve its split-off flow. The rest of the models can then be called in the order that they process the flows, as follows:

```
sp_1.s; gt_hp.c; mx_1.c; mx_2.c; ht_reac.c; hx_nz.h; nz_1.c;
```

Note that here the mixer models will use the flows previously saved by their secondary functions. Also, note that the hot side function is being called for the hx_nz model. The entire system configuration is thus represented by

```
gas_h2.c; pump_lp.c; pump_hp.c; hx_nz.c; sp_2.c; sp_1.c;
gt_lp.c; mx_1.s;   sp_2.s; mx_2.s;
sp_1.s; gt_hp.c; mx_1.c; mx_2.c; ht_reac.c; hx_nz.h; nz_1.c;
```

For each of the models used within the system one will need to define an instance of the model and to assign the appropriate parameter values. These are, of course, completely dependent on the problem. For example, the gas_h2 model instance and its parameter values might be defined as follows:

```
gas gas_h2 = {id="THR-tH2";  t=20;  p=1.29;  m=7.387;  v=200;};
```

Here we define the model with the name gas_h2 and then assign its flow id parameter the value "THR-tH2". The rest of the line then represents the chosen initial values for the temperature, pressure, mass flow rate, and velocity. The other models used would need similar declarations and are shown in the final inputs to the problem. As we are more concerned with showing how problems are set up, many of the input parameters are simply taken as their default values, and are not representative of an actual space propulsion system.

The rest of the inputs to this problem is formed by adding two additional function calls, one to print the gas flow output and one to print the model parameter output. We also include in this example a call to the power stack print function, pows.print. The final complete input for this problem is as follows:

```
modstack mods;
gas   gas_h2   = {id="THR-tH2";  t=20; p=1.29; m=7.387,  v=200;};
pump pump_lp  = {eff=0.67; pres=7.9;};
pump pump_hp  = {eff=0.81; pres=139.22;};
hx   hx_nz    = {t_cold=1050.0/1.8;};
sp   sp_2     = {sr=0.3;};
sp   sp_1     = {sr=0.3;};
gt   gt_lp    = {eff=0.23; pres=85.91;};
gt   gt_hp    = {eff=0.75; pres=85.91;};
ht   ht_reac  = {temp=5274/1.8;};
nz   nz_1     = {eff=0.85; pres=0.1;};
mx   mx_1, mx_2;

gas_h2.c;  pump_lp.c;  pump_hp.c;  hx_nz.c;  sp_2.c;  sp_1.c;
gt_lp.c;  mx_1.s;  sp_2.s;  mx_2.s;  sp_1.s;  gt_hp.c;
mx_1.c;  mx_2.c;  ht_reac.c;  hx_nz.h;  nz_1.c;
gass.print; mods.print; pows.print;
```

The outputs for this example, shown in the Appendix, are the result of the gass.print, mods.print, and pows.print function calls. The gass.print call displays the table of state points of exit flow from each model. All units in this table are SI with the exception of that for pressure which is atmospheres. Following the state-point outputs are the parameter outputsfor individual models, which were generated by the mods.print call. Finally, the table of model powers (input, loss, produced, and consumed) is generated by the pows.print function.

In looking at these outputs, one can see that if the pair of models gt_lp and pump_lp were meant to form a turbo-pump, then the power consumed by the pump would equal the power produced by the turbine, which is not the case here. The same situation holds with the gt_hp and pump_hp model pairs. Thus, it might be more appropriate to constrain the power produced and power consumed in these two model pairs. This is only an equation-solving task, similar to the second example. The first step is to determine what parameters could be varied to establish these constraints.

In general, many parameters can be varied within a system in order to establish constraints. The only criterion is that the constraints be functionally dependent on the chosen parameters. In this problem the most obvious parameters would be the pressure levels at the exit of the models concerned. For this problem, however, the pressure levels out of the models represent the pressure leading to the reactor and the nozzle and, thus, are important parameters of the problem. It would be best to fix these at the appropriate design level and vary some other parameters.

Another set of parameters might be the split ratios at the two splitters. Varying these split ratios alters the amounts of mass flow that can be directed to the two turbines generating more or less power. Using these split ratios, the vary statements would then look like the following:

```
vary(sp_2.sr, 0.3, 0.1, 0.9);
vary(sp_1.sr, 0.3, 0.1, 0.9);
```

where the starting value, lower bound, and upper bound were taken as 0.3, 0.1, and 0.9, respectively.

The constraints can be taken as

```
cons(sp_2.sr,  gt_lp.power.work+pump_lp.power.work);
cons(sp_1.sr,  gt_hp.power.work+pump_hp.power.work);
```

Here the constraint delimiters (the first argument) have been taken as the two split ratios and the actual constraint expressions as the sum of the power.work variables for the respective models. This variable is algebraic, with negative values meaning work consumed and positive values meaning work produced. Thus, the sum is used rather than a difference to equate work consumed with work produced.

Including the declaration of the task and adding these vary and cons statements to a task loop around the model calls are all that needs to be done to establish these system constraints. The new complete inputs are as follows:

```
modstack mods;
gas  gas_h2  = {id="THR-tH2";  t=20; p=1.29; m=7.387;  v=200;};
```

```
pump pump_lp = {eff=0.67; pres=7.9;};
pump pump_hp = {eff=0.81;  pres=139.22;};
hx   hx_nz   = {t_cold=1050.0/1.8;};
sp   sp_2    = {sr=0.3;};
sp   sp_1    = {sr=0.3;};
gt   gt_lp   = {eff=0.23; pres=85.91;};
gt   gt_hp   = {eff=0.75; pres=85.91;};
ht   ht_reac = {temp=5274/1.8;};
nz   nz_1    = {eff=0.85; pres=0.1;};
mx   mx_1, mx_2;
task a;

while (a.c)
  { vary(sp_2.sr, 0.3, 0.1, 0.9);
    vary(sp_1.sr, 0.3, 0.1, 0.9);
    gas_h2.c;  pump_lp.c;  pump_hp.c;  hx_nz.c;  sp_2.c;  sp_1.c;
    gt_lp.c;  mx_1.s;  sp_2.s;  mx_2.s;  sp_1.s;  gt_hp.c;
    cons(sp_2.sr,  gt_lp.power.work+pump_lp.power.work);
    cons(sp_1.sr,  gt_hp.power.work+pump_hp.power.work);
  }
mx_1.c;  mx_2.c;  ht_reac.c;  hx_nz.h;  nz_1.c;
gass.print; mods.print; pows.print;
```

Within these inputs, those models that appeared after the gas turbines which do not affect the constraints were not included within the task loop. This is only a computational performance issue, as all the models could be included if desired. In fact, the `gas_h2`, `pump_lp`, `pump_hp`, and `hx_nz` models could be put before the loop, since varying the split ratios will not affect any of their outputs. If that were done then, the system configuration and task loop would look like the following:

```
gas_h2.c;  pump_lp.c;  pump_hp.c;  hx_nz.c;
while (a.c)
  { vary(sp_2.sr, 0.3, 0.1, 0.9);
    vary(sp_1.sr, 0.3, 0.1, 0.9);
     sp_2.c; sp_1.c; gt_lp.c; mx_1.s; sp_2.s; mx_2.s; sp_1.s; gt_hp.c;
    cons(sp_2.sr,  gt_lp.power.work+pump_lp.power.work);
    cons(sp_1.sr,  gt_hp.power.work+pump_hp.power.work);
  }
mx_1.c;  mx_2.c;  ht_reac.c;  hx_nz.h;  nz_1.c;
```

When a task loop, such as the above, is placed around only a part of the system configuration, as defined by the model functional entries, the flows that must exist on the stacks at the start of each iteration through the task loop are correctly set by the `c` function of the task model controlling the loop. Thus, in this example, the `sp_2.c`, being the first model that is called within the loop, will always see the `hx_nz.c` model's flow on the stack even though the `gt_hp.c` model was the last model called during the previous iteration. In other words, as long as a task class is controlling the loop, the flow stacks will be correct. This is not the case, for example, if one were to simply put an iterative loop around some models using a `for`, `while`, or `do` loop.

The resulting output for this example is also shown in the Appendix. The only difference between this example and the previous one is the inclusion of the task loop iterations and the resulting changes within the mass flow rates through the system.

6.5 Coal-Fired Power Plant

This example involves a more realistic application for a coal-combustion steam power plant, schematically represented in Fig. 2. This power generation system includes two physical constraints, which will be discussed later. First, the model calls necessary to define the system configuration will be described.

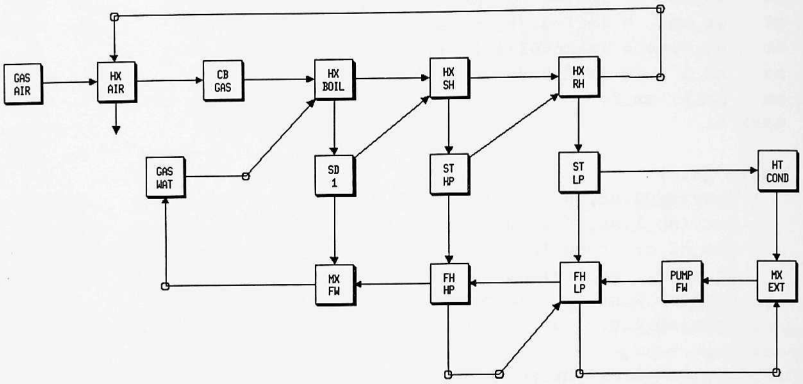


Figure 2. Coal-Fired Power Plant.

An air flow is first generated using an instance of the gas model, `gas_air`. The air flow is then passed through the cold side of an air heater, `hx_air`, followed by a combustor, `cb_gas`, where the coal is combusted. The products of combustion are then passed through the hot sides of three heat exchangers, denoted as `hx_boil`, for the steam boiler, `hx_sh`, for the superheater, and `hx_rh`, for the reheater. The combustion gases are then returned to the hot side of the air heater, `hx_air`, before being exhausted from the system. The complete air and combustion gas cycle can thus be modeled by the following sequence of model calls.

```
gas_air.c; hx_air.c; cb_gas.c; hx_boil.h; hx_sh.h; hx_rh.h; hx_air.h;
```

For the water-steam cycle, the flow path actually forms a closed loop. The method of handling such a loop is to "cut" the flow path at some point and initiate a flow at the cut point. Later, the flows entering and leaving that point are iterated until they are equal. To make this process easier, the `gas` model, which is used to initiate the flow, has an entry `cycl`, which represents the "back door" to the model. The `cycl` entry calculates the differences in the state variables between the flow entering the `cycl` entry and that leaving the `c` entry. One can then vary parameters within the system until these differences become zero. We will discuss how this is done in more detail later.

Additionally, one has to make a decision as to where to cut the flow path. To reduce the number of variables that one might have to iterate over, it is best to pick a point where some properties of the flow are known. For example, after a pump, the flow's pressure is known or after a heat exchanger, the flow's temperature may be known. Alternatively, a flow path might be cut at a point to reduce the number of other closed loops over which additional variables would need to be iterated. In the case of this steam plant, a number of closed loops actually exist due to the extraction of steam flow from the turbines leading into the feed water heaters. By cutting the flow path and initiating a water flow upstream of the steam boiler, however, only one cut point is needed. Thus, in this case we initiate a water flow at this point by using an instance of the gas model, `gas_wat`.

The water flow out of the `gas_wat` model is passed through the cold side of the boiler, `hx_boil.c`, followed by the steam drum, `sd_1c`. The water flow out of the steam drum is fed into the feed water mixer, `mx_fw.s`, and the steam from the drum, generated by the `sd_1.s` entry, is passed through the cold side of the superheater, `hx_sh.c`, followed by the high-pressure steam turbine, `st_hp.c`. The main portion of the steam is then passed through the reheater, `hx_rh.c`, then to the low-pressure steam turbine, `st_lp.c`, followed by the steam condenser, `ht_cond.c`, and then to the secondary side of the steam extraction mixture, `mx_ext.s`. That part of the steam cycle is represented as follows:

```
gas_wat.c; hx_boil.c; sd_1.c; mx_fw.s;
sd_1.s; hx_sh.c; st_hp.c; hx_rh.c; st_lp.c; ht_cond.c; mx_ext.s;
```

At this point, the secondary function of the high-pressure steam turbine model , st_hp.s, is called to retrieve any extraction steam, which is then passed to the hot side of the high-pressure feed water heater, fh_hp.h. The output flow from fh_hp.h, representing the drain cooler output, is then passed to the secondary side of the low-pressure feed water heater, fh_lp.s. Next, the extraction steam flow from the low-pressure steam turbine, st_lp.s, is fed into the hot side of the low-pressure feed water heater, fh_lp.h, and then mixed with the condensate from the condenser in mx_ext.c. The resulting feed water is then passed through the feed water pump, pump_fw.c. This part of the steam cycle is represented as follows:

```
st_hp.s; fh_hp.h; fh_lp.s; st_lp.s; fh_lp.h; mx_ext.c; pump_fw.c;
```

The water from the feed water pump is then passed through the cold sides of feed water heaters fh_lp and fw_hp , and then through mx_fw.c, and finally into the "back door" cycl entry of the gas_wat model, where the water-steam cycle started. This part of the cycle is represented as follows:

```
fh_lp.c; fh_hp.c; mx_fw.c; gas_wat.cycl;
```

The rest of the inputs to this example is formed by adding four function calls, one to print the gas flow output, one to print the gas composition, one to print the model parameter output, and one to print the power input/output.

The entire system configuration is, therefore, represented by the following sequence of calls to the different models.

For the air-gas cycle:

```
gas_air.c; hx_air.c; cb_gas.c; hx_boil.h; hx_sh.h; hx_rh.h; hx_air.h;
```

For the water-steam cycle:

```
gas_wat.c; hx_boil.c; sd_1.c; mx_fw.s;
sd_1.s; hx_sh.c; st_hp.c; hx_rh.c; st_lp.c; ht_cond.c; mx_ext.s;
st_hp.s; fh_hp.h; fh_lp.s; st_lp.s; fh_lp.h; mx_ext.c; pump_fw.c;
fh_lp.c; fh_hp.c; mx_fw.c; gas_wat.cycl;
```

For the output:

```
mods.print; gass.print; gass.mprint; pows.print;
```

As with the other examples, for each of the models used in the system, one will need to define an instance of the model and to define the appropriate parameter values. For example, in the combustor model, cb_gas, the coal composition is defined, in weight fractions, as 0.7793 carbon (carb), 0.054 hydrogen (h), 0.131 oxygen (o), 0.0073 sulfur (s), and 0.0744 water (h2o). The lower heating value and the mass flow rate of fuel are also defined by using lhv and mass. In the air flow initiator, the flow is identified as a "GAS" flow with a typical air composition, 79% nitrogen and 21% oxygen. For the water flow initiator, the flow is identified using "STM" , followed by the appropriate values for flow temperature, quality, pressure, mass flow rate, and velocity.

Two constraints are imposed on this system. First, the exhaust flow temperature of the products of combustion through the air heat exchanger is set to be 700 K. In general, many parameters can be varied within a system in order to satisfy this constraint. The only criterion is that the constraint be functionally dependent on the chosen parameter. In this problem, we take the parameter to be the combustor's (coal) mass flow rate, cb_gas.mass. The vary statement, in this case would then look like the following:

```
vary(cb_gas.mass, 15,12,25);
```

where the starting value, the lower bound, and upper bound are taken as 15, 12, and 25, respectively. The constraint in this case is written as:

```
cons(cb_gas.mass, hx_air.flh.t-700);
```

Here the constraint delimiter has been taken as the combustor mass flow rate and the actual constraint expression as the difference between the temperature of the hot side of the air heat exchanger, hx_air.flh.t, and the required temperature value of 700 K.

This task is achieved by including a definition of the task itself, here taken as b, and adding the vary and cons statements to a task loop around the air-gas cycle models, as will be shown later. The del parameter for this task has been set to 0.01, rather than using the default value, as the larger value tends to converge faster for this problem.

The second constraint is for the water loop closure. Assuming for simplicity that there are negligible pressure drops through the feed water heater, the pressure of the flow at the gas_wat initiator will be the same as at the feed water pump, which has been taken as 180 atm. Thus, the pressure of the initiated flow is known. However, the temperature of the flow is not known; thus, closing the water loop requires iterating over the initiated water temperature. It is better, however, to iterate over the value of the water's enthalpy, since enthalpy along with pressure will always uniquely define the state of water (note that temperature and pressure are related to each other in the two-phase region and, thus, are not independent variables). To do this without having to know reasonable enthalpy values, we use the gas model's option of inputting the water quality. Thus, within the gas_wat definition, we set t to be zero and then vary gas_wat.q in order to close the loop. Since the water should be subcooled at this point, we take quality to be between 0 and -2.5 starting at -0.5. The vary statement in this case would then look like the following:

```
vary(gas_wat.q, -0.5, -2.5, 0.0);
```

The value of the enthalpy difference (i.e., the difference in the water enthalpy between the start and end of the water-steam cycle) is contained within the variable gas_wat.d.h, so the constraint in this case can be written as:

```
cons(gas_wat.q, gas_wat.d.h);
```

Here again the constraint delimiter has been taken as the water quality.

This task is also achieved by including a definition of the task itself, here taken as a, and adding the vary and cons statements to a task loop around the water-steam cycle models, as shown below.

Following all the steps discussed, the complete input to this example is as follows:

```
modstack mods={conffile="tmp/splant.conf"; rdatfile="tmp/splant.rdat";};
gas    gas_air={id="GAS"; t=300; p=1.0; m=206.6979; v=20.0;
         comp[N2]=0.79; comp[O2]=0.21;};
cb     cb_gas={carb=0.7793; h=0.054; o=0.131; s=0.0073; h2o=0.0744;
         lhv=3.214e7; mass=20.22;};
gas    gas_wat={id="STM"; t=0.0; q=0.0; p=180; m=500; v=10;};
pump   pump_fw={eff=0.85; pres=180;};
sd     sd_1;
mx     mx_fw,mx_ext;
hx     hx_boil={q_cold=0.25;},
       hx_sh={t_cold=811;},
       hx_rh={t_cold=811;},
       hx_air={t_cold=600;};
gt     st_hp={eff=0.84; pres=50; ext=0.1;},
       st_lp={eff=0.86; pres=0.066; ext=0.1;};
fh     fh_hp, fh_lp;
ht     ht_cond={temp=0.0; qual=0.0;};
task   a, b={del=1e-2;};

while (a.c)
   {vary(gas_wat.q, -0.5, -2.5, 0.0);
    gas_wat.c; hx_boil.c; sd_1.c; mx_fw.s;
    sd_1.s; hx_sh.c; st_hp.c; hx_rh.c; st_lp.c; ht_cond.c; mx_ext.s;
    st_hp.s; fh_hp.h; fh_lp.s; st_lp.s; fh_lp.h; mx_ext.c; pump_fw.c;
    fh_lp.c; fh_hp.c; mx_fw.c; gas_wat.cycl;
    cons(gas_wat.q, gas_wat.d.h);
   }
while (b.c)
   {vary(cb_gas.mass, 15,12,25);
     gas_air.c; hx_air.c; cb_gas.c; hx_boil.h; hx_sh.h; hx_rh.h; hx_air.h;
    cons(cb_gas.mass, hx_air.flh.t-700);
   }
```

```
    mods.print; gass.print; gass.mprint; pows.print;
    mods.rdat;
```

Also added to this example was a call to the mods.rdat function. As discussed within the modstack class, this function will store on the rdatfile file the state-point information for displaying on the system diagrams (see Section 3). Thus, the mods parameters now include the file names, conffile, and the rdatfile. The resulting output for this example is shown in Appendix. As with the previous example, many of the input parameters of the models have simply been taken as their defaults, which are not representative of an actual coal-fired power plant. This results in inaccurate values for many of the component size parameters, such as, heat exchanger surface areas or weights. Again, our objective here is to show the steps in setting up problems.

6.6 PEM Fuel Cell System

In this example, we show a system making use of a PEM fuel cell for vehicle power. This example requires somewhat more sophistication on the part of the user than the previous ones in that some understanding of fuel cell systems is required. To make the system realistic, we designed it to run on methanol, making use of a steam reformer, and to operate at anode and cathode pressure levels of 2 atm and 3 atm, respectively. Thus, in addition to the pem model, a reformer, a compressor, a gas turbine, and a number of heat exchanger models will also be needed. Also, PEM fuel cells require tailoring of the input fuel flow to remove as much CO as possible; thus, the reformed fuel flow path will need some additional mixing with water to promote the water/gas shift reaction and some mixing with fresh air to promote the preferential oxidation of any remaining CO before entering the fuel cell. The PEM cell also needs to be cooled by a water flow that is supplied from a tank in which the water level is maintained. Thus, some scrubbing of the cathode and anode flows will be needed, with the extracted water being remixed with the coolant water loop. Additionally, some control needs to be placed on the reformer burner so that appropriate reformate temperature (as well as reformer catalyst temperatures) is maintained.

Figure 3 shows the PEM system configuration. The methanol fuel flow originates from a gas model, labeled as fuel, is pumped to the anode pressure level (pump_fuel), pre-heated (hx_preh), and then split into two flows (sp_fuel). The primary fuel flow goes into the reformer (form), and the secondary flow goes to a fuel/air mixer (mx_fuel) for use in the reformer burner. The reformate from the reformer is then passed through a mixer (mx_shif) representing the shift converter, another mixer (mx_prox) representing the preferential oxidizer, and a final flow cooler (hx_cool), and into the anode inlet entry of the PEM model, here labeled as pefc. The cathode flow starts from a gas model, denoted as air, is then compressed (cp_air) and split into two flows (sp_air), one for the reformer burner and the other for the main cathode flow. This main cathode flow is further split (sp_prox) to generate a flow for use in the preferential oxidizer (mx_prox), with the rest flowing into the cathode side of the pefc model.

The cathode exhaust from the fuel cell is first scrubbed of any water (sp_h2o) and then mixed with the reformer burner exhaust (mx_cath). The anode exhaust from the fuel cell is first compressed to the cathode pressure level (cp_anode), and then split into two flows in sp_anode. The first of these flows is mixed with additional air (mx_burn) and then with the extra fuel flow (mx_fuel) for use in the reformer burner. The rest of the flow from sp_anode is then mixed with the cathode/reformer burner flow (mx_anode), before it enters the gas turbine (gt_1) and a final flow condenser (cond_1). The condensate flow from the condenser is later mixed (mx_cond) with the coolant water loop.

The water loop starts at the water tank (water_tank), is pumped up to pressure (pump_water), cools the pefc, is mixed with water scrubbed from the cathode exhaust (mx_h2o), and is split (sp_wat) into two flows. The first of these is further split (sp_shif) to provide process water to the shift converter (mx_shif) and the secondary entry of the reformer (form). In forming the inputs to the system, this water flow will be needed before processing the fuel flow through the reformer; thus, the water loop is actually cut before the sp_shif model using another gas model labeled as wat. This water loop will then be closed by the appropriate system constraints. The rest of the water from the sp_wat model is used on the hot side of the fuel preheater (hx_preh), passed through the main heat rejection radiator (hx_rej), mixed with any condensate from the anode exhaust stream (mx_cond), and finally, reenters the water tank.

The only other components shown in the diagram are the coolant flows for the condenser, the main heat-rejection radiator, and the main-air compressor intercooler. Each of these minor flows is an air flow initiated using a gas model and then compressed using a cp model labeled as a fan.

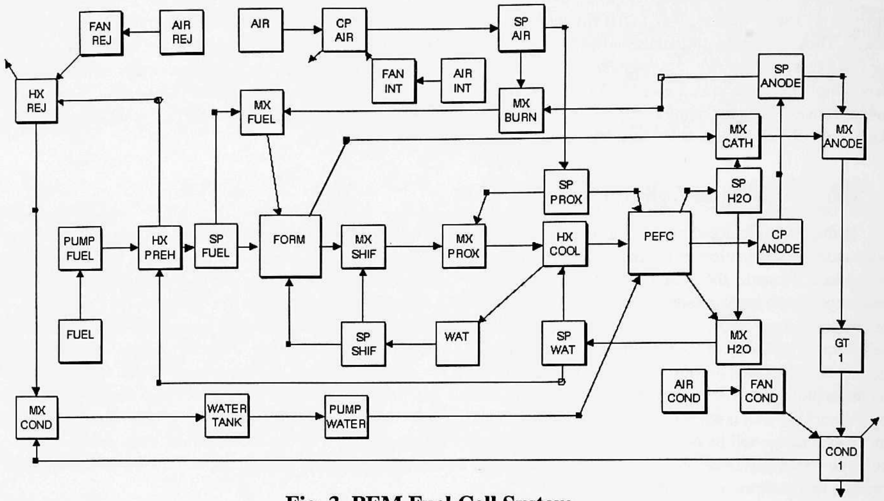


Fig. 3. PEM Fuel Cell System

In forming the system inputs, one starts with a component that generates a flow, following the flow path through those components processing this flow until a component is reached that terminates the flow. This is repeated for all flows, taking into account that some component functions need to be called before others, such as the mixer's secondary function before the mixer's calculational function. Using these rules, one can represent the above system configuration as follows:

```
air.c; cp_air.c; sp_air.c; sp_prox.c; mx_prox.s;
wat.c; sp_shif.c; form.s;
fuel.c;  pump_fuel.c; hx_preh.c; sp_fuel.c; form.c;  mx_shif.s;
sp_shif.s; mx_shif.c; mx_prox.c; hx_cool.h; pefc.ain;
sp_prox.s; pefc.c; sp_h2o.c; mx_cath.s;
sp_air.s; mx_burn.s;
sp_fuel.s; mx_fuel.s;
pefc.a; cp_anode.c; sp_anode.c; mx_burn.c; mx_fuel.c;
    form.h;  mx_cath.c; mx_anode.s;
sp_anode.s; mx_anode.c; gt_1.c; cond_1.c;
sp_h2o.s; h2o.cont; mx_h2o.s;
water_tank.c; pump_water.c; pefc.cool; mx_h2o.c;
    hx_preh.h; hx_rej.h; mx_cond.s;
cond_1.s; mx_cond.c; water_tank.cycl;
sp_wat.s; hx_cool.c; wat.cycl;
air_int.c; fan_int.c; cp_air.cool;
air_cond.c; fan_cond.c; cond_1.cool;
air_rej.c; fan_rej.c; hx_rej.c;
```

In the above each non-indented line represents one flow from its origin to its termination as defined by a model function not generating an output flow (the two indented lines are continuations of the previous lines). Thus, most of the lines start with either a gas model or a secondary entry of a sp model, and many of the lines end with a secondary entry of a mx model. The only exception for initiating flows is the cond_1.s function, which originates the condensate flow leaving the condenser.

As indicated above, the coolant flow loop was cut at the wat model, and thus, some system constraints will need to be applied to form the loop closure. Additionally, the flows of water into the water_tank model also form a loop which will require some closure. Starting with the water loop beginning and ending at the wat model, the only variables that need to be matched are the mass flow rate and temperature. The pressure will be whatever pressure the water pump has taken the flow to, neglecting the pressure drops through the fuel cell. Thus, one could vary the mass flow rate and temperature (or better yet, enthalpy) out of the wat.c function until they are equal to those entering the wat.cycl function. This will be done for the enthalpy using the gas model's option of specifying the flow's quality to define the model's exiting enthalpy. Thus, for the first constraint we have

```
vary(wat.q, 0.6, -0.2, 1.5);
cons(wat.q, wat.d.h);
```

For the mass flow rate, we can do something more direct. The mass flow rate generated by the wat model is used to supply process water to the reformer and the shift converter. Given a fuel flow rate, the amount of process water sufficient to provide the appropriate stoichiometry for these processes can be calculated once, and the wat.m parameter assigned this value. Then, since the sp_wat.c function is generating the flow that enters the wat.cycl function, we simply define the split ratio to give the flow rate

```
sp_wat.sr=wat.m/mx_h2o.fl.m;
```

where mx_h2o.fl.m is the flow rate into the splitter. If this statement is placed in the inputs after the mx_h2o.c function and before the sp_wat.c function, then the mass flow rates will be closed without use of the vary and cons functions. The actual value assigned to the wat.m parameter is, of course, dependent on how much process water one desires to include. The methanol reformer model does not specify this but generates outputs based on equilibrium chemistry and thermodynamics of the amounts of flows entering it. Generally, one uses 30%-50% excess water over fuel in a methanol/steam reforming process. Thus, if mrate is the molar flow rate of fuel, then the value of wat.m could be written as

```
wat.m=1.3*mrate*18.015*(1-sp_fuel.sr)/(1-sp_shif.sr);
```

taking into account that not all of the fuel is used in the reformer and not all of the water from the wat model is used in the reformer. If this appears complicated, one could add the vary/cons functions to close the water mass flows and then add another vary/cons, probably varying sp_wat.sr to constrain the appropriate process water stoichiometry.

The loop closure on the water tank also requires that the mass flow rates and temperatures be matched. However, the amount of mass flow rate within the whole water loop is really not defined by the loop closure. One could equally have loop closure for a large mass flow rate as for a small rate. Thus, the flow rate needs to be set by some other criteria. Since the water loop cools the fuel cell, we chose to vary the water mass flow rate such that the exit coolant flow from the cell is 5 K less than the cell temperature, which is taken as 353 K. Thus,

```
vary(water_tank.m, 1.1, 0.05, 2.0);
cons(water_tank.m, pefc.flcool.t-348);
```

Here, the starting value, lowe rbound, and upper bound are estimated and might have to be readjusted later to obtain a converged solution. One still needs to close the loop in terms of the temperature. Since the flows entering the tank come from a mixing of the condensate from the cond_1 model and from the hx_rej.h function, both of which have model parameters that permit specifying their exit temperatures, we chose to set these exit temperatures to the water_tank.t as follows:

```
hx_rej.t_hot=water_tank.t;
cond_1.texit=water_tank.t;
```

This will automatically close the loop in terms of temperature matching. However, even though the water mass flow rates have been set to a level for appropriate cooling of the fuel cell, it may still not be the case that the water entering the water tank is equal to that leaving. This is due to some of the coolant water being used as process water. Thus, one uses the condenser and the cathode scrubber to recover that process water. The amount that is recovered, however, will depend on the condenser's temperature. Thus, in order to ensure a closure on the mass flow rate, we vary the water tank's (and hence, condenser's) temperature,

```
vary(water_tank.t, 325, 305, 340);
```

```
cons(water_tank.t, water_tank.d.m);
```

The complete input for this problem with typical values for many of the input model parameters is as follows:

```
modstack mods;
gas    air={id="GAS"; t=300; p=1.0; m=4.989e-3*28.0; v=20.0;
            comp[O2]=0.21; comp[N2]=0.79; humid=0.5;};
gas     fuel={id="THR-CH4O"; t=300.0;  p=1.0; m=0.33e-3*32.042; v=20.0;
            comp[CH3OH]=1.0; frozen=1;};
gas    air_cond={id="GAS"; t=300; p=1.0; m=1.0; v=5.0;
            comp[O2]=0.21; comp[N2]=0.79; humid=0.5;};
gas    air_rej={id="GAS"; t=300; p=1.0; m=5.0; v=5.0;
            comp[O2]=0.21; comp[N2]=0.79; humid=0.5;};
gas    air_int={id="GAS"; t=300; p=1.0; m=1.0; v=5.0;
            comp[O2]=0.21; comp[N2]=0.79; humid=0.5;};
pump   pump_fuel={pres=3.0; eff=0.75;};
pump   pump_water={pres=2.0; eff=0.75;};
gas    wat={id="STM"; t=0.0; q=0.45; p=2.0; m=0.4e-3*18; v=5.0;
            comp[H2O]=1.0;};
gas    water_tank={id="STM"; t=323; p=1.0; m=0.627; v=5.0;
            comp[H2O]=1.0;};
gas    h2o={id="STM"; comp[H2O]=1.0;};
hx     hx_preh={t_cold=343; ufc=30; ufh=50; thickwall=1e-4;};
hx     hx_rej={t_hot=323; ufc=30; ufh=50; thickwall=1e-4;};
hx     hx_cool={t_hot=353; ufc=30; ufh=50; thickwall=1e-4;};
sp     sp_air={sr=1.333/5.0;};
sp     sp_prox={sr=1.0-0.063/3.656;};
sp     sp_wat;
sp     sp_shif={sr=0.0256/1.0256;};
sp     sp_fuel={sr=0.1;};
sp     sp_anode={sr=0.0;};
sp     sp_h2o={sr=-1; ssr[H2Oc]=1.0;};
mx      mx_burn, mx_fuel,  mx_cath, mx_shif, mx_prox, mx_cond, mx_anode,
          mx_h2o;
cond   cond_1={texit=323.0; u=30.0; thick=5e-4;};
gt     gt_1={mode="d"; pres=1.0; eff=0.80;};
cp     cp_air={mode="d"; pres=3.0; eff=0.80; nstages=2;};
cp     fan_cond={pres=1.005; eff=0.80;};
cp     fan_int={pres=1.005; eff=0.80;};
cp     fan_rej={pres=1.005; eff=0.80;};
cp     cp_anode={pres=3.0; eff=0.80;};
reform form={texit=473.15;};
pem    pefc={mode="d"; option='t'; curden=0.575;  celltemp=353;
         fuelutil=0.85;};
task   task_1={acc=1e-3; prt=2; del=-1e-5; maxit=20;};

gass.noform[CH4]=1; gass.noform[CH3OH]=1; gass.noform[C8H18]=1;

while (task_1.c)
   {vary(wat.q,0.6,-0.20,1.5);
    vary(water_tank.m, 1.1, 0.05,2.0);
    vary(water_tank.t, 325, 305, 340);
```

```
        fuel.m=0.33e-3*32.042;
        air.m=5.0e-3*28.0;
        wat.m=0.33e-3*1.3*18.0153*(1-sp_fuel.sr)/(1.0-sp_shif.sr);
        cond_1.texit=water_tank.t;
        hx_rej.t_hot=water_tank.t;

        air.c; cp_air.c; sp_air.c;
        sp_prox.c; mx_prox.s;
        wat.c; sp_shif.c; form.s;
        fuel.c;  pump_fuel.c; hx_preh.c; sp_fuel.c; form.c;  mx_shif.s;
        sp_shif.s; mx_shif.c; mx_prox.c; hx_cool.h; pefc.ain;
        sp_prox.s; pefc.c; sp_h2o.c; mx_cath.s;
        sp_air.s; mx_burn.s;
        sp_fuel.s; mx_fuel.s;
        pefc.a; cp_anode.c;
        sp_anode.c; mx_burn.c; mx_fuel.c;
        form.h;  mx_cath.c; mx_anode.s;
        sp_anode.s; mx_anode.c; gt_1.c; cond_1.c;
        sp_h2o.s; h2o.cont; mx_h2o.s;
        water_tank.c; pump_water.c; pefc.cool; mx_h2o.c;
        sp_wat.sr=wat.m/mx_h2o.fl.m; sp_wat.c;
        hx_preh.h; hx_rej.h; mx_cond.s;
        cond_1.s; mx_cond.c; water_tank.cycl;
        sp_wat.s; hx_cool.c; wat.cycl;
        air_int.c; fan_int.c; cp_air.cool;
        air_cond.c; fan_cond.c; cond_1.cool;
        air_rej.c; fan_rej.c; hx_rej.c;
        pows.c;

        cons(wat.q,wat.d.h/wat.fl.h);
        cons(water_tank.m, pefc.flcool.t-348.0);
        cons(water_tank.t, water_tank.d.m);
       }
     mods.print; gass.print; gass.mprint; pows.print;
```

In these inputs we have also turned off the formation of methane and methanol within the gas property code. This was done with the `gass.noform`[CH4]=`gass.noform`[CH3OH]=`gass.noform`[C8H18]=1 line. The reason for this step is that one would expect that these species would not form due to the slow reaction kinetics, but would thermodynamically form from equilibrium chemistry alone, which is what the gas property code employs. Also we have defined the amount of fuel and air as expressions within the task loop in terms of molar flow rate and molecular weights. The outputs generated by the above inputs are given in Appendix.

7.0 Graphics

The graphics currently available within GCtool are simple but are sufficient to generate reasonable looking two- and three-dimensional plots.

7.1 Two-dimensional Plot (plot) Model Class

Two-dimensional plots of user-selected independent (x values) and dependent (y values) variables are generated by using a model class denoted as `plot`. For each plot desired, an instance of this plot class should be defined. When the plot is generated, a new window will pop open on the screen to display the curves. At present, up to eight curves can be generated within each plot.

The variables that can be defined for the plot model are as follows:

title[256] -	Character string that is up to 256 characters long, representing the title to the plot. Input. For each occurrence of '\n' within the title string, a new line will be generated within the title. The title lines are centered above the plot. The default font used on the plots is Helvetica-Bold. The default plot window when opened is designed to accommodate only one line of title, the window will need to be resized to see all the title lines, if more than one.
titleps -	Point size for the title characters (14). Input.
labps -	Point size used for the axis labels and legends (12). Input.
hid -	Height of the plot window when first popped open (4.5 inches). Input.
wid -	Width of the plot window when first popped open (5.0 inches). Input.
width -	Width of the plot area in inches (3.5). Input.
hidth -	Height of the plot area in inches (2.5). Input.
mode[8] -	Character string indicating the type of scales to be used on the plot. At present, the following values are recognized: "linear" for both x- and y-axis being linear scales, "xlog" for the x-axis being a log scale, "ylog" for the y-axis being a log scale, and "loglog" for both axes being log scales.
xlab[48] -	Character string representing the x-axis label (""). Input.
xmin -	Lower bound of the independent variables (0.0). Input.
xmax -	Upper bound of the independent variables (1.0). Input.
xinc -	Increment to be used for labeled x-axis values (0.2). Input. The value `xinc` is only used for linear scale plots. For the log scale modes, only the `xmin` and `xmax` values are used to determine the scale, and in this case `xmin` should not be zero.
xpos -	x position in inches from the lower left-hand corner of the plot window (1.0). Input. Note that this value is increased by 1.0 inch when the print menu button is pressed for producing plots on the printer. The default values for `xpos` and `ypos` were chosen so that the initial plot window will show a complete plot when initially popped open without too much empty space within the window.
xtic -	Number of tick marks between x increment values (5). Input. For log scales this value is ignored as the tick marks are calculated to represent the nonlinear log scale.
xleg -	Location in inches from the origin where the left side of the legend is to appear (4.0). Input. The legend location defaults to the right side of the plot. Initially, when the plot window opens, the legend may be too far to the right to appear within the window. The window can be easily resized to see the legend. At the time the window is opened, it is not known whether or not the plot will actually have a legend; thus, the plot window has been kept small in order to accommodate just the plot.
xnote -	x location in inches from the origin where a character string note is to be written.
ylab[48] -	Character string representing the y-axis label (""). Input.
ymin -	Lower bound of the dependent variables (0.0). Input.
ymax -	Upper bound of the dependent variables (1.0). Input.

yinc -	Increment to be used for the labeled y-axis values (0.2). Input. The same comments made about the x-axis concerning log scales are also true for the y-axis.
ypos -	y position in inches from the lower left-hand corner of the plot window (0.75). Input. Like the `xpos` value, this too is increased by an inch when the print menu button is pressed.
ytic -	Number of tick marks between y increment values (5). Input.
yleg -	Location in inches from the origin where the top side of the legend is to appear (1.75). Input.
ynote -	y location in inches from the origin where a character string note is to be written.
grid -	Flag indicating whether or not a background grid should be displayed on the plot. One turns on the grid and zero turns it off (1). Input.
rot -	Angle of rotation in degrees in the counterclockwise direction that the entire plot will make with the horizontal (0.0). This parameter is only effective for plots when printed on the printer.
selfscale -	Flag indicating whether or not the plot should make use of self-scaling (1). See the discussion on self-scaling below. Input.
leg[8][32] -	Character strings used as the legend for each curve on the plot.
note[1024] -	Character string value used to display a note on the plot.

The data for each plot are obtained by using the `c` function for the model. This function requires three arguments consisting of the curve number from 0 to 7, representing one of up to eight curves on each plot and the x,y pairs of data to be plotted. Thus, one would write

```
    plot_1.c(0,x,y);
```

to plot the x,y pair in the 0-th curve within `plot_1`. The plots generated use straight line segments between the plotted points. Multiple curves can be generated by extending the input array with additional curve numbers, and x,y pairs as follows:

```
    plot_1.c(0,x,y, 1,x,z, 2,x,t);
```

To provide the legends for each curve, the `leg` array can be assigned character string values representing the legend for each curve within the plot. The legend character strings are preceded by a short length of the type of line (solid, dash, etc.) used to represent the curve. Each curve, by default, has a slightly different type of line. The location of the legend can be adjusted by using the `xleg` and `yleg` values.

The note capability was provided so that brief, but possibly multi-line, notes could be written somewhere on the plot. The parameter `note` takes a single character string representing the note to be displayed. As with the plot's title string, a new line is generated for each occurrence of the '\n' character within the string. The upper left corner of the note is at the location (`xnote,ynote`).

At present, there is a little delay between popping open a window and continuing the execution of the GC input. Thus, since the act of popping open a window may take some time, it is possible for very simple problems that the entire GC input may have been executed before the plot window has been opened. No data are lost in this case, as the data going to the plot window are stored and simply plotted when the window becomes open. At present, only 400 x,y pairs are stored per curve. A check is made when doing the plotting that the new x,y pair is at least one pixel different from the previous x,y pair. Thus, 400 values are usually sufficient for most plots. Data are also properly stored if a plot window is closed. If damage has been done to the plot that does not get automatically repaired, then a slight resizing of the window will usually fix the damage.

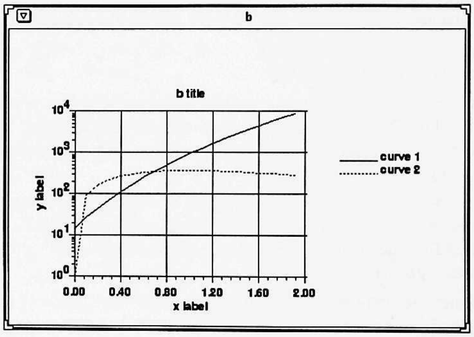
As the plot windows are based on the OpenLook windows, these plot windows can be moved, resized, closed, and opened. The resizing, however, does not resize the plot itself. The plot can be resized by changing the plot parameters, `width` and `hidth`, and then using the `axis` function. Any damage to the window is automatically repaired from the data stored for the plot. Plot windows will not be terminated until the GCtool session is terminated or the user explicitly terminates the plot by using its window frame menu item, `quit`.

The following GC input is an example of the use of the plot class:

```
   plot  a={title="a long title \nwith a second line \nand a third",
              xlab="x label", ylab="y", xmax=2.0, xinc=0.4, ymax=200,
        yinc=50};
        plot  b={title="b title", xlab="x label", ylab="y label", mode="ylog",
                    xmax=2, xinc=0.4, ymin=1.0, ymax=10000, yinc=1000
                    leg={"curve 1", "curve 2"} };
        plot  c={title="c title", xlab="x stuff", ylab="y stuff",
                    xmin=1, xmax=100, ymin=1.0, ymax=1000, mode="loglog"};
        double x,y,z;
        for (x=0.0; x<=2.0; x+=0.1)
           {y=10*exp(x+1); z=1000*x*exp(-x)+1.0;
             a.c(0,x,y);    b.c(0,x,15*pow(x+1.0,6.0)); b.c(1,x,z);
           }
        for (x=1.0; x<=100; x+=5.0)
           c.c(0,x,5*x);
```

Here three plot windows will be popped open. Window "a" is a plot of $10e^{(x+1)}$ versus x from 0 to 2.0. In window "b" two curves are generated on a semi-log plot, the first of $15(x+1)^6$ versus x and the second of $1000xe^{-x}+1$ versus x. In window "c" a plot of 5x versus x is plotted on a log-log scale. The "b" plot window is shown below.

Besides the `c` functions, there is an `axis` function that can redraw the plot. This function is useful as new values for the plots parameters can be defined, and then when this function is called, the plots will reflect these new parameter values. For example, after executing the above input, one could change the scale values or even go from linear to log-log and then replot the data without rerunning the example.



The background of the plot window has one menu item when the right mouse button is pressed. This item is labeled "print" and will send the plot to the laser printer and generate a file labeled as the plot name concatenated with ".tmp". This file contains the postscript code for the plot and can be manually edited or inserted into other documents.

To generate plots without too much input being required for their setup, a self-scale option is also provided. This option works in several different ways, depending on the value of the `selfscale` parameter and only provides self-scaling for linear axis. A zero value turns off all self-scaling. With a non-zero value for `selfscale`, the minimum, increment, and maximum axis parameters are determined as the plot data are obtained.

The value of the `selfscale` parameter determines how and also when the plot axis are changed. As the data for the plot are obtained, the plot may need to replot all the previous data with new scales to accommodate the incoming new data. To prevent this replotting for each new data point, the `selfscale` parameter can be given a negative value. After all the data have been obtained, `selfscale` can then be set to a positive value and replotted by calling the `axis` function.

The absolute value of the selfscale parameter is used to determine how the bounds and increments are to be calculated. A selfscale value of 1 (or -1) indicates that any input bounds are to be used if possible. Note that this does not preclude an automatic readjusting of these bounds to accommodate a reasonable increment value, even if the data do not exceed the specified bounds. For example, if the lower bound were set at -1 and the upper at +1 but the incoming data to be plotted went from 0 to 50, the lower bound would also be adjusted lower so that each increment would be the same. Of course, if no bounds were specified, the default values will be the starting point for the bounds. A selfscale value of 2 (or -2) will basically set all of the initial bounds to zero, ignoring any input values. This has the effect that if all the data are of the same sign, zero will be one of the bounds. For many plots this is often what is desired; however, if all the data are between, say, 800 and 900, then one may want to give the axis a lower bound of 800 and use selfscale of one.

7.2 Three-dimensional Plot (plot3) Model Class

The plot3 model is used to generate three-dimensional plots within a pop-up window of a function z=z(x,y). The model has two functional entries. The first denoted as data is used to store the data for the surface. This entry is called first with two arguments, nx and ny (in double precision), representing the number of x intervals and the number of y intervals. The number of data points along each axis is one more than the number of intervals. The data are assumed to be equally spaced along each axis from xmin to xmax and ymin to ymax. The subsequent calls to data require three arguments, i, j, and $z_{i,j}$, where i and j are indices along the x and y axis from 0 to nx and 0 to ny, respectively. Thus, the data entry is called many times. Once the data have been defined, the second entry, c, is called to actually make the plot. As with the two-dimensional plots, a number of additional parameters can be assigned to provide the plot size and window size, as well as to rotate and tilt the plot to any viewpoint.

The variables that can be defined for the plot3 model are as follows:

title[256] -	Character string that is up to 256 characters long, representing the title to the plot. Input. For each occurrence of '\n' within the title string a new line will be generated within the title. The title lines are centered above the plot. The default font used on the plots is Helvetica-Bold. The default plot window when opened is designed to accommodate only one line of title; the window will need to be resized to see all the title lines, if more than one.
titleps -	Point size for the title characters (14). Input.
labps -	Point size used for any notes (12). Input.
hid -	Height of the plot window when first popped open (4.5 inches). Input.
wid -	Width of the plot window when first popped open (5.0 inches). Input.
width -	Width of the plot area in inches (4.0). Input.
hidth -	Height of the plot area in inches (4.0). Input.
xlab[48] -	Character string giving the x-axis label ("X"). Input.
xmin -	Lower bound of the x variables (0.0). Input.
xmax -	Upper bound of the x variables (1.0). Input.
xnote -	x location in inches from the origin where a character string note is to be written (1.0). Input.
ylab[48] -	Character string giving the y-axis label ("Y"). Input.
ymin -	Lower bound of the y variables (0.0). Input.
ymax -	Upper bound of the y variables (1.0). Input.
zlab[48] -	Character string giving the z-axis label ("Z"). Input.
zmin -	Lower bound of the z variable. This is obtained from the input data.
zmax -	Upper bound of the z variable.
rota -	Angle of rotation in degrees in the counterclockwise direction that the entire plot will make with the x-axis (45.0). A zero angle places the x-axis directly to the left in the plot.
tilt -	Angle of rotation in degrees that the view point makes with the horizontal plane (30.0). Input. A zero angle places the view point in the x-y axis plane.

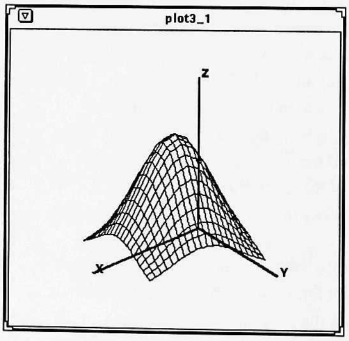
nx -	Number of x intervals. The x values are equally spaced between `xmin` and `xmax`.
ny -	Number of y intervals. the y values are equally spaced between `ymin` and `ymax`.
note[1024] -	Character string value used to display a note on the plot (""). Input.
hide -	Flag to remove (1) all hidden lines from the surface or keep (0) the hidden lines (1). Input.
xlines -	The number of lines in the x-direction used to display the surface (20). Input.
ylines -	The number of lines in the y-direction used to display the surface (20). Input. If both `xlines` and `ylines` are zero, the surface will not be displayed.
dx -	Increment between data values within the x direction.
dy -	Increment between data values within the y direction. Note that after the `data` function has been called to convey the number of x and y intervals, `dx` and `dy` are defined by using the current setting for `xmin`, `xmax`, `ymin`, and `ymax`.

As an example of a three-dimensional plot, we present the following:

```
    plot3 plot3_1={xmin=-2, xmax=2, ymin=-2, ymax=2, tilt=20, rota=30,
        xlines=20,  ylines=20};
    double i,j,x,y,z;

    plot3_1.data(10,10);
    for (i=0; i<=10; i++)
      {x=plot3_1.xmin+i*plot3_1.dx;
        for (j=0; j<=10; j++)
          {y=plot3_1.ymin+j*plot3_1.dy;
           z=exp(-(x*x+y*y)/3);
           plot3_1.data(i,j,z);
          }
      }
    plot3_1.c;
```

This will generate a plot window as follows:



8.0 Interfacing with Precompiled Models

8.1 Introduction

For fast execution of a system problem, it is best to precompile the component models using a conventional C compiler rather than to interpret them. This requires linking the models to the GC interpreter. For GC this is quite simple and consists of defining an `init` function for the model, including the `init` function in a special `clinker` procedure, and adding the model's data structure (enhance with size and off-set information) to the usual header file included with the GC inputs. The details of the model interfacing will be shown by way of an example.

8.2 Model Structures and Functions

A model, or more specifically, a model class, in GC is nothing more than C data structure and a collection of functions. As an example of a model, suppose that we have a generic heat exchanger that includes two functions that process the hot and cold fluid flows of the model. These functions will be denoted as `h()` and `c()`, respectively. The structure of the model might be as follows (here we will not actually show the coding of the model equations, only the overall structure):

```
struct hx
    {char name[16],config[16];
     double heat,... ;
     CFUNC init, h, c;
    };
int hx_c(z)
    struct hx *z;
    {....}
int hx_h(z)
    struct hx *z;
    {...}
int hx_init(args)
    char *args[];
    {struct hx *z;
     z=(struct hx*)args[0];
     z->c=hx_c; z->h=hx_h;
     z->heat=0.0;
     strcpy(z->config,"c;1 0,p h;1 0,p");
     strcpy(z->name,args[1]);
     return sizeof *z;
    }
```

In addition to the flow processing entries, a model must also have an `init` function to correctly interface with GC. The `init` function is used to define initial values for the elements of the model's data structure. The functions of the model are referred to within the data structure as pointers by using the `CFUNC` type. Note that the CFUNC type was defined as a pointer to a function returning an integer; thus, all of the model functions that are to be called directly from within the GC inputs need to be of that type. These pointers are used to locate the functions when referred to within the GC inputs and must be assigned values within the `init` function. Thus, the C functions that are executed when the `hx` members `hx.c` and `hx.h` are called within the GC inputs are defined within the `hx_init` function as `hx_c` and `hx_h`, respectively. Note that the actual function names do not have to match the name referred to by the `CFUNC` type, but it is useful to have some convention; for example, we always name the function as the model name, followed by an '_' and the function name defined within the model structure.

As explained in Section 2, such member functions always take as their first argument a pointer to the structure of which they are a member. In the case of the `init` member function, this pointer is the first element of an `args` array. The second element of the `args` array is a character string containing the name of the structure as used within the GC inputs. Generally, this second argument is stored within a `char` array within the structure (`name` in this example) and can be used for labelling the model variables during printout. Besides assigning the model function pointers to their appropriate functions, the `init` function should also return the size of the model's structure. These two things are technically the only requirements for the `init` function, although it is useful to include the initialization of any other model parameters, such as the initialization of the `name` parameter, the `heat` parameter, and the `config` parameter in the above example. The `config` parameter will be discussed below.

While this simple example uses only two member functions (not counting the `init` function), a model may have any number of additional member functions. Each should be defined within the model's structure as a `CFUNC` type and given a value within the `init` function if it is to be called from the GC interpreter. For example, most models will have a `print` entry for printing out the model parameters. When models have a function defined as `print`, they will automatically be called whenever the `mods.print` function is called. Additionally, some models may need to allocate additional variables (which should be referenced by pointers within the model structure). These variables would need to be freed when the model is deleted by the GC interpreter. This can be accomplished by defining a termination function denoted as `term`. This function, if it exists for a model, is automatically called by the GC interpreter just before it deletes the model structure. This function only requires as an argument the pointer to the model structure. Note that the model structure itself is both allocated and freed by the GC interpreter and never within one of the model's functions.

8.3 Linking C Functions to the GC Interpreter

The linking of the model function names and the actual procedure that is called is accomplished within the model's `init` function, as discussed above. However, the model `init` functions themselves must be located so that they may be called to perform this linking process. Additionally, other functions may be defined that are not member functions of some model and that need to be called from within the GC inputs. An auxiliary function, denoted as `clinker`, linkes a function name as used in the GC inputs to the actual precompiled function that is called. `Clinker` takes as an argument a character string name of the function and returns the pointer to the function that will be called. This pointer is then saved by the GC interpreter and used to locate the function when it is called. Thus, `clinker` links precompiled functions to their character string representations and must be supplied to the GC interpreter as an external function. `Clinker` and the model header file, to be described below, are the only links between the GC interpreter and the models. In the case of the `init` functions, the argument to `clinker` is the model structure type name followed by ".init". For example, in the `hx` model above, this would be a string such as `"hx.init"`. Thus, given `"hx.init"`, `clinker` would return a pointer to the `hx_init` function.

As an example of a simple `clinker` function, we present the following:

```
void* clinker(char *name)
  {if (strcmp("hx.init",name)==0) return (void*)hx_init;
   else if (strcmp("task.init",name)==0) return (void*)task_init;
     ...
   else return (void*)0;
  }
```

If the `clinker` does not recognize the input name, it should return a null pointer. In this way the GC interpreter is informed that an unknown function is being called, and it can put out an appropriate message to the user. Note that for the above code to compile, each of the model functions would need to be declared. Alternatively, `clinker` could call secondary linker functions, one for each model file, placed within the model files themselves, and then only these secondary linker functions would need to be declared. As an example,

```
extern void *clinker_mod0(), *clinker_mod1(), *clinker_mod2();
void* clinker(name)
  char *name;
  {void *ptr;
   if ((ptr=clinker_mod0(name))!=0)
     return ptr;
   else if ((ptr=clinker_mod1(name))!=0)
     return ptr;
   else
```

```
        return clinker_mod2(name);
    }
```

Besides the model `init` functions, other C functions can also be included. Thus, if one has some function, say `xfunc`, that is not a member of some C structure, it may also be included by simply including the line,

```
    else if (strcmp("xfunc",name)==0) return (void*)xfunc;
```

within the `clinker` function. Note that in this case `xfunc` would also need to be declared within the GC inputs as

```
    CFUNC xfunc;
```

which could be put into the interface file, as discussed below.

8.4 Generating Model Header Files

As was mentioned within the Section 2, the layout in computer memory of compiled model structures might differ from the layout imposed by the GC interpreter. Thus, for structures that might be passed to precompiled functions, one should include the layout information within the structures themselves. As this could be difficult to do by hand, an additional code, designated as GCintf can be used to generate this information. GCintf takes as inputs a list of model source or header files and extracts all of the structures. It then generates a code that, when compiled and executed, will generate an interface file that includes the information needed by the GC interpreter. In practice, this procedure is automated within a makefile, so that one needs only type "make intf" to generate the header file. The following is an example of the makefile.

```
    intf:
            gcintf ../mod0.h mod1.c sub/mod2.c clinker.c >gctemp.c
            cc -o gctemp gctemp.c
            gctemp >intf.h
            rm gctemp.c gctemp
```

Here one model header file (`../mod.h`) and two model files (`mod1.c` and `sub/mod2.c`) in several directories along with the `clinker.c` file will be scanned to locate the model structures and to generate the file `gctemp.c`. This last file, which contains the coding necessary to generate the header file, is then compiled and executed to generate the new header file. In listing the models or other files that need to be scanned, the files should be put in the order that any files containing model structures that depend on the structures within another file are listed after the other file. If the gcintf code cannot make sense of the dependencies, the `gctemp.c` file itself will simply list the error and the file the in which error occurred, as well as the structures that were scanned.

This interface file, here denoted as "intf.h", is nothing more than a standard header file of structure definitions but with the additional offset and size information that will be interpreted by the GC interpreter. This header file can be included within the GC inputs, defining all of the model types that are available for use in the system simulations. When solving problems with GCtool, the interface file is automatically included and does not have to be explicitly included within the inputs.

The "intf.h" file may also include other information for the GC code to properly handle the models. For example, as discussed in the Section 5, most models require several stacks, such as the `gass` and `shfts`, to be defined before they are used. Such additional declarations can also be added to the interface file by using the special `/*/` and `*/` comment delimiter and the word `INTERFACE` at the end of a model's file. Only the last of these special interface comments is actually used; thus, a good place to put them is in the `clinker.c` file and include it as one of the files to be scanned. As an example, to include these stacks within "intf.h" for use by GC one would put the following comment at the end of the `clinker.c` file that is fed to the GCintf generator:

```
    /*/ INTERFACE
    gasstack gass;
    shftstack shfts;
    */
```

Lines between `/*/ INTERFACE` and `*/` will simply be copied exactly as they appear into the "intf.h" file. This would be the place to include the declarations of other variables or functions, such as the `xfunc` defined in the previous section so that they are also known within the GC inputs.

The `intf.h` file that is generated should correspond to the model library that is linked to GCtool. So that GCtool can locate this interface file, one additional external variable besides the `clinker` function is required. This variable, denoted as `interface`, is defined as follows:

```
        char *interface="/.../intf.h"
```

where ... refers to any other directory information necessary to locate `intf.h`. While this definition of interface can be put within any of the model files that are linked to GCtool, a good place for this declaration is within the `clin-ker.c` file.

8.5 Model Configuration Parameter

The `config` model parameter, while not actually necessary for running a model, is necessary for the model to be properly displayed within a system's diagram when the GCtool's `draw` button is pressed. This model parameter gives information to GCtool concerning which model functions processes which flows. The information is supplied as a character string containing a set of entries for each model function that process a flow. This information is laid out within the string as follows. First the model function name is listed, followed directly by a semicolon and the number of flows processed by this model function. Then, for each of the flows processed by this function, the flow number, a comma, and either 'p', 'i', or 'o' is specified. The flow number corresponds to the array elements used in the `mods.sysflows` array. Thus, presently, 0 stands for the `gastype` flows, and 1 stands for the `shfttype` flows. The 'p', 'i', or 'o' letter is then used to identify whether the flow is a pass-through flow (i.e., one that both enters and exits the model), and input flow (one that enters but does not exit a model) or an output flow (one that only exits a model). Each of these flow specifications should be separated from each other by at least one blank.

As an example, take the heat exchanger example above. Since the `c` and `h` functions each process only a single gastype flow we have `config`="c;1 0,p h;1 0,p". As another example, the gas turbine model discussed previously (Section 5), has `c`, `s`, and `shft` functional entries. The `shft` function processes a `shfttype` flow as an input, the `s` function has only a single `gastype` as an output flow, and the `c` function processes both a `gastype` as a pass-through flow and a `shfttype` as an output flow. Thus, in this case, `config`="c;2 0,p 1,o s;1 0,o shft;1 1,i".

8.6 Summary of Model Interfacing

In summary, the details of interfacing a model with GCtool consist of the following.

- Develop an `init` function for the model, in which all of the pointers to the model member functions are defined, as are any initial values to model parameters (including the `config` parameter), and the size of the model's C structure is returned.
- Add the model's `init` function to the clinker procedure.
- Add any special variables that the model needs, such as flow stacks, to the interface file, usually through the use of the special `/*/ INTERFACE` comment with the clinker's source file.
- Define the name of the interface file by using the global `interface` variable.
- Remake the `intf.h` header file by using the GCintf code.

Usually, when adding a new model to an existing collection, no additional stacks or variables are needed within the INTERFACE comment and the `interface` variable, itself, will probably not have been changed; thus, only the first, second, and last of the above need to be done.

8.7 Additional Interfacing Information

The previous sections have defined the steps necessary to interface a model with the GC interpreter; however, they have not defined those aspects of using a model to represent some component within a system. Thus, one needs to develop models that interact with other models by means of the flows between them. The reason that this has not been discussed is that the GC code is independent of the flows that are used within a system. All aspects of the flow manipulations are handled within the models. Thus, it is possible to replace the existing method of handling flows with some other technique. However, the stack-based method discussed below has worked very well and the tools for using it are all available within the GC package. In addition, the models will generally need to make calls to property procedures and possibly to various mathematical utilities. The GC package provides user interfaces to a number of property and utility procedures.

8.7.1 Stacks and Flows

As indicated within the Section 5, a flow is defined as nothing other than a C structure. This structure can be as large or as small as is needed to convey the information between the component models. All of the flows within a system are stored as substructures within the model structures. Thus, for example, the heat exchanger model might have two flows stored within its data structure representing the fluid flows on the hot and cold sides. When a model is called, a mechanism is needed for obtaining a flow from the previous model or making a flow known to any succeeding models. This mechanism is based on stacks. Thus, for each flow type there is a stack upon which the flows or, more properly, the address of the flows is placed. A collection of functions has been defined for placing flows on the stack, removing flow from the stack, iterating over all the flows on the stack, etc. These functions all start with the word `stack` and are defined as follows:

stack_in -	Used to initialize a stack and takes a pointer to the stack's data structure and a character string. The character string should be a name for the stack. This function is used to set the pointers to the top, bottom, and current stack elements to NULL, indicating that the stack is initially empty.
stack_put -	Used to put a new item on the stack. It takes three arguments. The first is a pointer to the stack's data structure to define which stack is to be used. The second is a pointer (typed cast to a `void*` pointer) to the item that is to be stored on the stack. The last argument is also a `void*` pointer and is used to define an alternative pointer variable used to locate this item. If this pointer is NULL, then the item will be found on the stack, using the second argument. The use of this alternative pointer will become clearer after the discussion of the `stack_find` and `stack_fname` functions. This `stack_put` function will always attempt to locate the item on the stack before attempting to place it on the stack again. If it is found, a current item pointer is reset to that element. This current item pointer is used by the `stack_get` function. If the item was not found on the stack, the item is put on top of the stack, and the current item pointer is reset to this item.
stack_get -	Used to retrieve from the stack the item pointed to by the current item pointer. It takes only one argument, a pointer to the stack's data structure, and returns the pointer (typed cast to a `void*` pointer) of the item. The current item pointer is readjusted to the previous item on the stack before returning. Note that the current item pointer will cycle to the end of the stack once the first element is retrieved. That is, the element previous to the first element is taken as the last element, giving rise to a circular stack.
stack_find -	Used to find an item on the stack. It takes two arguments, the first being the pointer to the stack's data structure, and the second being an input pointer associated with the item to be found. If the item was originally put onto the stack using the alternative pointer (see `stack_put` above), then the pointer to the item associated with this alternative pointer is returned if the input pointer is the same as this alternative pointer. In this way, the alternative pointer is just another way of labeling the item. If the item was placed on the stack with no alternative pointer, then the input pointer is assumed to be that of an item on the stack and, if found, returns the pointer to that item. In either case, if the item is not found on the stack, a NULL pointer is returned.
stack_fname -	Used to find an item on the stack by means of a character string name. In this case, the items must be placed on the stack using the alternative pointer (see `stack_put` above), which, in this case, must be a pointer to a character string (still typed cast to a `void*` pointer). `Stack_fname` takes two arguments, the first being the pointer to the stack's data structure and the second being a character string of the item's name to be found. If found, the pointer to the item is returned; if not, a NULL pointer is returned.

stack_next -	Used to iteratively cycle over all items on a stack. It takes two arguments: the first is the pointer to the stack's data structure, and the second is an address of a pointer to a `stacklink` structure. This structure is used to hold the stack together in a linked list. Initially, this second argument should point to a NULL value, and thereafter will be reassigned by the `stack_next` function. Basically, this second argument is used as a counter to locate the next item on the stack. Each call to `stack_next` will either return the pointer to an item, or if no more items are on the stack, a NULL pointer.
stack_del -	Takes two arguments, a pointer to this stack's data structure and a pointer to an item. The stack is searched for the item and, if found, is deleted from the stack. Note that since only pointers to the items and not the actual items are stored on the stack, deletion of an item from the stack only deletes the links that hold the item to the stack and does not delete the actual item.
stack_term -	Takes only one argument, a pointer to the stack's data structure, and deletes the entire stack. As with the `stack_del` function, it is only the stack and not the actually items that are deleted.

These functions permit one to easily handle the passage of flows between the models. Before exiting a model that generates a flow, the `stack_put` function is called to place the flow on a stack, and on entry to a model that requires a flow, the `stack_get` function is called to retrieve a flow from the stack. For example, the `gastype` flows are all placed on a stack whose address is denoted as `gass`. If within a model this gas flow is denoted as `z->fl`, where `z` is the address of the model structure containing this flow, the call

```
stack_put(gass, (void*)&z->fl, 0);
```

would place the flow onto the `gass` to be retrieved by the next model. Similarly, the call

```
z->fl=*(struct gasstype*)stack_get(gass);
```

would retrieve the last flow placed on the stack and assign it to `z->fl`. Additionally, one can iterate over all values that have been placed on the stack, such as in printing them out, by using the `stack_next` function. In this case one needs to define a pointer to the `stacklink` structure to iterate through the stack. For example, the code fragment

```
struct gastype *gp; struct stacklink *l=0;
while (gp=(struct gastype*)stack_next(gass,&l)
  ...;
```

would sequentially return a pointer, `gp`, to each `gastype` flow placed on `gass`.

8.7.2 Property Functions

A number of property functions are available for use with a model's `gastype` flows. These are used for calculating the thermodynamic properties (transport properties are not currently available) for a flow given the flow's pressure and one of either temperature, enthalpy, or entropy. If these functions are called in lieu of individual coding within a model for calculating the properties, a more self-consistent thermodynamics can be obtained for a systems analysis problem.

The first of these functions is `prop` and is the calculational procedure for the general thermodynamci properties. For any instance of the `gastype` class, `prop` can be called to determine the thermodynamic properties of the flow either as a function of `p` and `t`, `p` and `h`, or `p` and `s` by using as its second argument the letter 't', 'h', or 's', respectively. The first argument of `prop` is the address of the flow. If the name of the flow is `fl` and is pointed to by the model's structure pointer labeled as z, the call to the property code would look like,

```
prop(&z->fl, 't');
```

This would calculate the values of all of the flow's thermodynamic parameters (Section 5.2.1), `h, s, r, q`, and for `id="GAS"`, also the `comp` array as a function of the flow's `t` and `p`. The calls where 't' is replaced by 'h' or 's' would be similar.

Another function, `sat`, is used to determine the saturation properties of the flow at the flow's pressure. This function only returns values for flows with the "THR" or "STM" `id`'s as the "GAS" flows are not considered condensable, although they may contain liquid water or methanol, and the "LIQ" flows are purely liquids and, thus, do not have the vapor saturation properties included. If called with a "GAS" or "LIQ" flow, an error message is displayed and the run is terminated. `Sat` requires four arguments, the first is the address of the flow, and the rest are pointers to double-precision variables representing the returned values of critical pressure (atm), the saturation liquid enthalpy, and the saturation vapor enthalpy (J/kg). A call to `sat` looks as follows:

```
        sat(&z->fl, &pc, &hl, &hs);
```

where `&z->fl` is a pointer to the flow, and `pc` is the critical pressure, and `hl` and `hs` are the saturation enthalpies. Note that if one needs the saturation temperature for a given pressure, a call would have to be made to `prop` after `sat` is called.

Another function, `atom`, is only needed for flows with the "GAS" id and is used to calculate the kg-atom/kg of flow of the individual atoms making up the flow. This function requires one argument of the flow's address and uses the flow's `comp` array to determine the values of the flow's `atoms` array. A call to `atom` would look like

```
      atom(&z->fl);
```

The `atoms` array should be updated whenever a "GAS" type flow undergoes a change in which chemical species are either added or removed from the flow. Note that for "GAS" type flows it is the `atoms` array that actually determines the chemical make-up of the flow. This array remains constant until the flow either has new species added to it or removed from it. The flow's `comp` array, on the other hand, will change just like the flow's temperature or pressure and reflects only the current values of the species moles per kilogram of flow.

8.7.3 Mathematical Utilities

In addition to making use of the GCtool mathematical utilities (equation solver, optimizer, and integrator) from within the GCtool inputs, one can also call these utilities from within the models. In this section, we consider the interfaces to these utilities.

Each of the mathematical utilities is iterative and, thus, requires some sort of iterative loop. When defining some task that is localized to a model, this loop would be coded within the model itself. It is possible that the loop would be coded outside of the model, such as that used by the `dyn` tasks for the dynamic models. Here we show examples of the iterative loops within the models themselves.

For the solution of a set of algebraic equations, the typical structure for solving the problem is as follows:

```
        struct task task1;
        char *args[2];
        args[0]=(char*)&task1; args[1]="task1";
        task_init(args);
        while (task_c(&task1))
          {  vary(&x1, s1, lb1, ub1, &task1);
             vary(&x2, s2, lb2, ub2, &task1);
                       .    .    .
             cons(&x1, f1, &task1);
             cons(&x2, f2, &task1);
                      .    .    .
          }
        task_term(&task1);
```

Here a `struct task` is declared, denoted as `task1`. The `task_init` function is then called to obtain the default values for many of the task parameters, such as `acc`, `del`, and `maxit`. As described above, all model `init` functions take an argument array with the structure address and its name as the elements of the array. Alternatively, the initial values of the various task parameters, `acc`, `del`, `maxit`, etc., can be defined by using assignment statements, for example, `task1.acc=1e-3`. A `while` loop is then started to define the iterations over the task body. The `vary` and `cons` functions are also similar to their use in GCtool, only here, the addresses of the variables need to be specified The other arguments to the `vary` function are the starting value, lower bound, and upper bound. One additional argument is required by the `vary` function, which is not present using the GC `vary` operator and is simply a pointer to the task class for this problem, which, in this case, was defined as `task1`. The `cons` function arguments are the address of some variable; this is just the delimiter for the constraint, the equation residual, and again the pointer to the task structure.

For optimizations, the same task class generation and loop structure as with the equation-solving task are generated. Only now, in addition to the `vary` and `cons` functions, the `icons` function can be used to define inequality constraints, such as

```
      icons(&xn,   fn,   &task1);
```

The `mini` function must be called to define the objective function, `obj`, to be minimized, such as

```
      mini(obj, &task1);
```

Again, these functions are similar to those used in the GC inputs but with the first arguments replaced by the addresses of the variables and with an additional argument consisting of the task pointer. Of course, for optimization problems the number of equality constraints as specified by the `cons` functions should be fewer than the number of variables. Note that, at present, optimization tasks should not be nested; thus, the use of an optimization task within a model would preclude use of the model within an optimization task set up within the GC inputs.

For integrating a set of first-order differential equations, again the same task class generation and loop structure as the above are used; however, the only function that can appear within the loop is the function `diff`. This function is used to define the dependent variable being integrated and its derivative value for each value of the independent variable, which is denoted as `task1.time`. For example,

```
     diff(&x, dxdt, &task1);
```

Here `dxdt` would be defined within the loop as the derivative `x` with respect to time (i.e., the right-hand side of the differential equation). The initial values for the dependent variables being integrated would be assigned before the loop is started. As with the use of integration tasks within GC, the integration loops cannot be nested.

The final call to `task_term` in the above is used to clean up any variables used by the task in solving the problem.

References

1. H. Geyer, "GPS Language and Utility Classes Documentation," unpublished report (1994).

2. H. Geyer, "GPSTool User's Manual," unpublished report (1994).

3. *PostScript Language Reference Manual*, Addison-Wesley Publishing, New York (1987).

4. M. J. D. Powell, "A Hybrid Method for Nonlinear Equations," in *Numerical Methods for Nonlinear Algebraic Equations*, Gordon and Breach Science Publishers, New York (1970).

5. M. J. D. Powell, "A Fast Algorithm for Nonlinearly Constrained Calculations," 1977 Dundee Conference on Numerical Analysis, Dundee, United Kingdom (1977).

6. C.W. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice Hall, Englewood Cliffs, NJ (1971).

7. J. C. Amphlett, M. J. Evans, R. A. Jones, R. F. Mann, and R. D. Weir, "Hydrogen Production by the Catalytic Steam Reforming of Methanol. Part 1: The Thermodynamics," Can. J. Chem. Eng. 59, 720-727 (1981).

8. J. C. Amphlett, M. J. Evans, R. F. Mann, and R. D. Weir, "Hydrogen Production by the Catalytic Steam Reforming of Methanol. Part 1: Kinetics of Methanol Decomposition Using Girdler G66B Catalyst," Can. J. Chem. Eng. 63, 605-611 (1985).

9. J. C. Amphlett, R. F. Mann, and R. D. Weir, "Hydrogen Production by the Catalytic Steam Reforming of Methanol. Part 3: Kinetics of Methanol Decomposition Using C18HC Catalyst," Can. J. Chem. Eng. 66, 950-956 (1988).

10. J. C. Amphlett, R. F. Mann, C. McKnight, and R. D. Weir, "Production of Hydrogen Rich Gas by Steam Reforming of Methanol over Copper Oxide-Zinc Oxide Catalysts," Proc. 20th IECEC, Miami Beach, FL, 772-780 (1985).

11. C. J. Jiang, D. L. Trimm, and M. S. Wainwright, "Kinetic Study of Steam Reforming of Methanol over Copper-Based Catalysts," Appl. Catal. A: General 93, 245-255 (1993).

Appendix. Outputs from Examples

Use of Vary and Cons to Solve a Single Equation

```
task: a  n=0  f=6.321206e-01
  x= 1.000000e+00
  c= 6.321206e-01
  h= 7.1266e-02 hs= 7.1266e-02 mu=0.00e+00 n=7.13e-02 s=7.13e-02 a=1.00e+00

task: a  n=1  f=5.690847e-02
  x= 7.330436e-01
  c= 5.690847e-02
  h= 5.7761e-04 hs= 5.7761e-04 mu=0.00e+00 n=6.98e-04 s=6.98e-04 a=1.00e+00

task: a  n=2  f=6.026605e-03
  x= 7.066324e-01
  c= 6.026605e-03
  h= 6.4778e-06 hs= 6.4778e-06 mu=0.00e+00 n=9.79e-06 s=9.79e-06 a=1.00e+00

task: a  n=3  f=6.978994e-05
  x= 7.035041e-01
  c= 6.978994e-05
```

Use of Multiple Vary's and Cons's to Solve a System of Equations

```
task: a  n=0  f=3.174881e+00
  x= 2.000000e+00  2.000000e+00  2.000000e+00
  c=-1.000000e+00 -2.253856e+00  2.000000e+00
  h= 2.1370e+00 hs= 2.1370e+00 mu=7.61e+00 n=4.18e+02 s=1.44e-01 a=8.89e-02

task: a  n=1  f=3.134960e+00
  x= 2.061622e+00  2.153135e+00  2.008621e+00
  c=-1.026095e+00 -2.209666e+00  1.972937e+00
  h= 2.0799e+00 hs= 2.0799e+00 mu=6.61e+00 n=2.92e+02 s=1.37e-01 a=1.08e-01

task: a  n=2  f=3.060456e+00
  x= 2.163727e+00  2.411827e+00  2.021546e+00
  c=-1.057566e+00 -2.133150e+00  1.922919e+00
  h= 1.9770e+00 hs= 1.9770e+00 mu=5.61e+00 n=1.41e+02 s=1.28e-01 a=1.58e-01

task: a  n=3  f=2.927054e+00
  x= 2.299266e+00  2.787530e+00  2.032131e+00
  c=-1.099439e+00 -2.002226e+00  1.830292e+00
  h= 1.8034e+00 hs= 1.8034e+00 mu=4.61e+00 n=6.94e+01 s=1.17e-01 a=2.30e-01

task: a  n=4  f=2.693334e+00
  x= 2.481947e+00  3.336961e+00  2.035323e+00
  c=-1.140796e+00 -1.787473e+00  1.660593e+00
  h= 1.5273e+00 hs= 1.5273e+00 mu=3.61e+00 n=3.41e+01 s=1.06e-01 a=3.37e-01

task: a  n=5  f=2.289259e+00
  x= 2.733308e+00  4.135351e+00  2.022142e+00
  c=-1.130996e+00 -1.457221e+00  1.355751e+00
  h= 1.1190e+00 hs= 1.1190e+00 mu=2.61e+00 n=1.46e+01 s=1.14e-01 a=5.11e-01

task: a  n=6  f=1.607120e+00
  x= 3.066868e+00  5.237910e+00  1.972048e+00
  c=-9.659670e-01 -9.868557e-01  8.221050e-01
  h= 5.8934e-01 hs= 5.8934e-01 mu=1.61e+00 n=4.59e+00 s=2.31e-01 a=7.51e-01

task: a  n=7  f=5.876752e-01
  x= 3.493864e+00  6.634444e+00  1.856100e+00
  c=-4.150844e-01 -4.131462e-01 -4.875719e-02
  h= 9.8227e-02 hs= 9.8227e-02 mu=6.10e-01 n=4.56e-01 s=6.49e-02 a=9.44e-01

task: a  n=8  f=8.432075e-02
  x= 3.722838e+00  7.487833e+00  1.919058e+00
  c=-7.398842e-02 -5.598153e-03 -4.005451e-02
  h= 1.4789e-03 hs= 1.4789e-03 mu=0.00e+00 n=3.53e-03 s=2.17e-04 a=5.50e-01

task: a  n=9  f=5.275489e-02
  x= 3.759556e+00  7.562479e+00  1.938630e+00
  c= 5.267110e-02 -2.687419e-03 -1.269695e-03
  h= 6.9599e-04 hs= 6.9599e-04 mu=0.00e+00 n=5.06e-04 s=1.40e-04 a=7.00e-01

task: a  n=10  f=5.944556e-03
  x= 3.743895e+00  7.534873e+00  1.934776e+00
  c=-5.912006e-03  3.103616e-04 -5.381605e-04
  h= 8.7871e-06 hs= 8.7871e-06 mu=0.00e+00 n=5.20e-06 s=1.75e-06 a=7.38e-01

task: a  n=11  f=6.240470e-05
  x= 3.745464e+00  7.537629e+00  1.935315e+00
  c=-5.920530e-05  2.378631e-06 -1.958115e-05
x=3.75 y=7.54 z=1.94
```

<u>Use of Multiple System Tasks</u>

```
task: b  n=0  f=2.000000e+00
  x= 2.000000e+00
  c= 2.000000e+00
  h= 2.5000e-01 hs= 2.5000e-01 mu=0.00e+00 n=2.50e-01 s=2.50e-01 a=1.00e+00

task: b  n=1  f=2.500001e-01
  x= 1.500000e+00
  c= 2.500001e-01
  h= 3.9063e-03 hs= 3.9063e-03 mu=0.00e+00 n=5.10e-03 s=5.10e-03 a=1.00e+00

task: b  n=2  f=4.081634e-02
  x= 1.428571e+00
  c= 4.081634e-02
  h= 1.0412e-04 hs= 1.0412e-04 mu=0.00e+00 n=1.94e-04 s=1.94e-04 a=1.00e+00

task: b  n=3  f=1.189769e-03
  x= 1.414634e+00
  c= 1.189769e-03
  h= 8.8472e-08 hs= 8.8472e-08 mu=0.00e+00 n=1.75e-07 s=1.75e-07 a=1.00e+00

task: b  n=4  f=6.007310e-06
  x= 1.414216e+00
  c= 6.007310e-06

task: a  n=0  f=2.465739e+00
  x= 2.000000e+00  2.000000e+00
  c=-1.000000e+00 -2.253856e+00

task: b  n=0  f=5.807310e-06
```

```
  x= 1.414216e+00
  c= 5.807310e-06

task: b  n=0  f=6.007310e-06
  x= 1.414216e+00
  c= 6.007310e-06
  h= 1.8870e+00 hs= 1.8870e+00 mu=7.76e+00 n=4.09e+02 s=9.30e-02 a=8.15e-02

task: b  n=0  f=5.551528e-02
  x= 1.414216e+00
  c=-5.551528e-02
  h= 3.8524e-04 hs= 3.8524e-04 mu=0.00e+00 n=3.85e-04 s=3.85e-04 a=1.00e+00

task: b  n=1  f=3.852393e-04
  x= 1.433843e+00
  c= 3.852393e-04

task: a  n=1  f=2.440552e+00
  x= 2.055521e+00  2.135376e+00
  c=-1.021250e+00 -2.216605e+00
  h= 1.8440e+00 hs= 1.8440e+00 mu=6.76e+00 n=2.95e+02 s=8.68e-02 a=9.73e-02

task: b  n=0  f=9.321621e-02
  x= 1.433843e+00
  c=-9.321621e-02
  h= 1.0566e-03 hs= 1.0566e-03 mu=0.00e+00 n=1.06e-03 s=1.06e-03 a=1.00e+00

task: b  n=1  f=1.056617e-03
  x= 1.466349e+00
  c= 1.056617e-03
  h= 1.3576e-07 hs= 1.3576e-07 mu=0.00e+00 n=1.33e-07 s=1.33e-07 a=1.00e+00

task: b  n=2  f=1.170996e-05
  x= 1.465985e+00
  c=-1.170996e-05

task: a  n=2  f=2.392801e+00
  x= 2.149123e+00  2.366971e+00
  c=-1.046488e+00 -2.151826e+00
  h= 1.7659e+00 hs= 1.7659e+00 mu=5.76e+00 n=1.49e+02 s=7.87e-02 a=1.39e-01

task: b  n=0  f=1.272847e-01
  x= 1.465985e+00
  c=-1.272847e-01
  h= 1.8847e-03 hs= 1.8847e-03 mu=0.00e+00 n=1.88e-03 s=1.88e-03 a=1.00e+00

task: b  n=1  f=1.884656e-03
  x= 1.509397e+00
  c= 1.884656e-03
  h= 4.1319e-07 hs= 4.1319e-07 mu=0.00e+00 n=4.01e-07 s=4.01e-07 a=1.00e+00

task: b  n=2  f=2.709710e-05
  x= 1.508764e+00
  c=-2.709710e-05

task: a  n=3  f=2.308036e+00
  x= 2.276396e+00  2.707661e+00
  c=-1.078475e+00 -2.040569e+00
  h= 1.6326e+00 hs= 1.6326e+00 mu=4.76e+00 n=7.47e+01 s=6.68e-02 a=2.02e-01
```

```
task: b  n=0  f=1.721483e-01
  x= 1.508764e+00
  c=-1.721483e-01
  h= 3.2546e-03 hs= 3.2546e-03 mu=0.00e+00 n=3.25e-03 s=3.25e-03 a=1.00e+00

task: b  n=1  f=3.254629e-03
  x= 1.565813e+00
  c= 3.254629e-03
  h= 1.1633e-06 hs= 1.1633e-06 mu=0.00e+00 n=1.12e-06 s=1.12e-06 a=1.00e+00

task: b  n=2  f=5.926978e-05
  x= 1.564755e+00
  c=-5.926978e-05

task: a  n=4  f=2.162196e+00
  x= 2.448517e+00  3.205920e+00
  c=-1.107719e+00 -1.856893e+00
  h= 1.4179e+00 hs= 1.4179e+00 mu=3.76e+00 n=3.72e+01 s=5.17e-02 a=2.97e-01

task: b  n=0  f=2.354908e-01
  x= 1.564755e+00
  c=-2.354908e-01
  h= 5.6623e-03 hs= 5.6623e-03 mu=0.00e+00 n=5.66e-03 s=5.66e-03 a=1.00e+00

task: b  n=1  f=5.662317e-03
  x= 1.640003e+00
  c= 5.662317e-03
  h= 3.2737e-06 hs= 3.2737e-06 mu=0.00e+00 n=3.12e-06 s=3.12e-06 a=1.00e+00

task: b  n=2  f=1.298307e-04
  x= 1.638236e+00
  c=-1.298307e-04

task: a  n=5  f=1.914216e+00
  x= 2.683948e+00  3.928191e+00
  c=-1.092509e+00 -1.571829e+00
  h= 1.0946e+00 hs= 1.0946e+00 mu=2.76e+00 n=1.67e+01 s=3.83e-02 a=4.54e-01

task: b  n=0  f=3.081942e-01
  x= 1.638236e+00
  c=-3.081942e-01
  h= 8.8478e-03 hs= 8.8478e-03 mu=0.00e+00 n=8.85e-03 s=8.85e-03 a=1.00e+00

task: b  n=1  f=8.847793e-03
  x= 1.732299e+00
  c= 8.847793e-03
  h= 7.2922e-06 hs= 7.2922e-06 mu=0.00e+00 n=6.89e-06 s=6.89e-06 a=1.00e+00

task: b  n=2  f=2.400278e-04
  x= 1.729674e+00
  c=-2.400278e-04

task: a  n=6  f=1.501885e+00
  x= 2.992013e+00  4.921391e+00
  c=-9.532758e-01 -1.160570e+00
  h= 6.6122e-01 hs= 6.6122e-01 mu=1.76e+00 n=5.89e+00 s=3.66e-02 a=6.89e-01

task: b  n=0  f=3.611914e-01
  x= 1.729674e+00
  c=-3.611914e-01
  h= 1.0901e-02 hs= 1.0901e-02 mu=0.00e+00 n=1.09e-02 s=1.09e-02 a=1.00e+00

task: b  n=1  f=1.090148e-02
  x= 1.834084e+00
  c= 1.090148e-02
  h= 9.9308e-06 hs= 9.9308e-06 mu=0.00e+00 n=9.36e-06 s=9.36e-06 a=1.00e+00

task: b  n=2  f=3.100318e-04
  x= 1.831025e+00
  c=-3.100318e-04

task: a  n=7  f=8.755773e-01
  x= 3.352964e+00  6.155649e+00
  c=-6.192090e-01 -6.190443e-01
  h= 2.1934e-01 hs= 2.1934e-01 mu=7.60e-01 n=1.22e+00 s=1.03e-01 a=9.34e-01

task: b  n=0  f=3.515875e-01
  x= 1.831025e+00
  c=-3.515875e-01
  h= 9.2176e-03 hs= 9.2176e-03 mu=0.00e+00 n=9.22e-03 s=9.22e-03 a=1.00e+00

task: b  n=1  f=9.217585e-03
  x= 1.927034e+00
  c= 9.217585e-03
  h= 6.3356e-06 hs= 6.3356e-06 mu=0.00e+00 n=6.02e-06 s=6.02e-06 a=1.00e+00

task: b  n=2  f=2.294686e-04
  x= 1.924581e+00
  c=-2.294686e-04

task: a  n=8  f=1.033512e-01
  x= 3.704242e+00  7.396947e+00
  c=-8.402441e-02 -6.017784e-02
  h= 2.9320e-03 hs= 2.9320e-03 mu=0.00e+00 n=1.06e-02 s=2.74e-03 a=9.62e-01

task: b  n=0  f=5.086342e-02
  x= 1.924581e+00
  c=-5.086342e-02
  h= 1.7461e-04 hs= 1.7461e-04 mu=0.00e+00 n=1.75e-04 s=1.75e-04 a=1.00e+00

task: b  n=1  f=1.746113e-04
  x= 1.937795e+00
  c= 1.746113e-04

task: a  n=9  f=3.471529e-02
  x= 3.754875e+00  7.554651e+00
  c= 3.468833e-02 -1.367775e-03
  h= 3.0142e-04 hs= 3.0142e-04 mu=0.00e+00 n=3.75e-04 s=5.94e-05 a=6.08e-01

task: b  n=0  f=1.376106e-02
  x= 1.937795e+00
  c= 1.376106e-02
  h= 1.2607e-05 hs= 1.2607e-05 mu=0.00e+00 n=1.26e-05 s=1.26e-05 a=1.00e+00

task: b  n=1  f=1.260816e-05
  x= 1.934244e+00
  c= 1.260816e-05

task: a  n=10  f=1.567039e-02
  x= 3.741289e+00  7.530313e+00
```

```
    c=-1.564774e-02  8.422847e-04
    h= 6.1442e-05 hs= 6.1442e-05 mu=0.00e+00 n=3.56e-05 s=1.26e-05 a=7.45e-01

task: b  n=0  f=4.218462e-03
    x= 1.934244e+00
    c=-4.218462e-03
    h= 1.1891e-06 hs= 1.1891e-06 mu=0.00e+00 n=1.19e-06 s=1.19e-06 a=1.00e+00

task: b  n=1  f=1.188908e-06
    x= 1.935335e+00
    c= 1.188908e-06

task: a  n=11  f=1.530847e-04
    x= 3.745520e+00  7.537728e+00
    c= 1.527830e-04 -9.606582e-06
x=3.75 y=7.54 z=1.94
```

Use of Icons and Mini to Solve an Optimization Problem

```
task a  n=1  meq=1  f= 6.0257e+01
    x= 1.0000e+00  2.0000e+00  3.0000e+00
    c=-1.0000e+00 -2.0000e+00
    l= 4.7355e+02

task a  n=2  meq=1  f= 4.9995e-01
    x= 1.5000e+00  1.5000e+00  0.0000e+00
    c=-4.9998e-05  1.5000e+00
    l= 9.9991e-05

task a  n=3  meq=1  f= 4.9995e-01
    x= 1.5000e+00  1.5000e+00  0.0000e+00
    c=-4.9998e-05  1.5000e+00
x=1.50 y=1.50 z=0.00
```

Use of Diff to Solve a System of Differential Equations

```
time=0.00 x=1.000e+00 y=2.000e+00 z=0.000e+00
time=1.00 x=3.680e-01 y=3.298e+00 z=-1.964e+00
time=2.00 x=1.355e-01 y=5.438e+00 z=-6.011e+00
time=3.00 x=4.983e-02 y=8.966e+00 z=-1.298e+01
time=4.00 x=1.826e-02 y=1.478e+01 z=-2.458e+01
time=5.00 x=6.717e-03 y=2.438e+01 z=-4.376e+01
```

Gas Turbine System

```
thermodynamic data for HYDROGEN with flow id = THR-tH2
     pc=12.800000, tc=33.200000, tb=20.400000, molwt=2.016000

                         output of model gastype flows

  model          temp    pres   mass        enth       entr     dens     vol      qual
                  (K)    (atm)  (kg/s)      (J/kg)   (J/kg-K) (kg/m^3) (m^3/s)

gas1            300.0    1.00 1.000e+00  2.499e+04  1.32e+04 8.19e-02 1.22e+01  1.00
cp1             544.4    6.00 1.000e+00  3.376e+06  1.39e+04 2.70e-01 3.70e+00  1.00
ht1            1000.0    6.00 1.000e+00  9.976e+06  2.27e+04 1.47e-01 6.79e+00  1.00
gt1             666.4    1.00 1.000e+00  5.097e+06  2.42e+04 3.69e-02 2.71e+01  1.00
```

```
                     output of model parameters

gas1         id=THR-tH2  area=0.000e+00 dt=0.00e+00 dp=0.00e+00 dm=0.00e+00 dh=0.00e+00
cp1           rpm=1.000e+00  power=-3.351e+06  heat=0.000e+00  nstages=1
              rat_cm=1.7321e+01  rat_crpm=5.7735e-02  rat_pr=6.0000  rat_eff=0.8800
              cm=1.0000e+00  crpm=1.0000e+00  pr=6.0000  eff=0.8800
ht1           heat=6.5995e+06
gt1           rpm=1.000e+00  power=4.879e+06
              rat_cm=5.2705e+00  rat_crpm=3.1623e-02  rat_pr=6.0000  rat_eff=0.8400
              cm=1.0000e+00  crpm=1.0000e+00  pr=6.0000  eff=0.8400
```

<u>Gas Turbine System with Fixed Net Power Constraint</u>

```
task: a  n=0  f=3.847218e+07
  x= 1.000000e+00
  c=-3.847218e+07
  h= 6.3409e+02 hs= 6.3409e+02 mu=0.00e+00 n=6.34e+02 s=6.34e+02 a=1.00e+00

task: a  n=1  f=3.962498e-01
  x= 2.618112e+01
  c= 3.962498e-01
  h= 6.7266e-14 hs= 6.7266e-14 mu=0.00e+00 n=6.73e-14 s=6.73e-14 a=1.00e+00

task: a  n=2  f=1.490116e-08
  x= 2.618112e+01
  c=-1.490116e-08

                       output of model gastype flows

  model            temp     pres   mass        enth        entr     dens      vol      qual
                    (K)     (atm)  (kg/s)      (J/kg)     (J/kg-K) (kg/m^3)  (m^3/s)

gas1               300.0    1.00 2.618e+01  2.499e+04  1.32e+04 8.19e-02 3.20e+02  1.00
cp1                544.4    6.00 2.618e+01  3.376e+06  1.39e+04 2.70e-01 9.68e+01  1.00
ht1               1000.0    6.00 2.618e+01  9.976e+06  2.27e+04 1.47e-01 1.78e+02  1.00
gt1                666.4    1.00 2.618e+01  5.097e+06  2.42e+04 3.69e-02 7.10e+02  1.00


                     output of model parameters

gas1          id=THR-tH2 area=0.000e+00 dt=0.00e+00 dp=0.00e+00 dm=0.00e+00 dh=0.00e+00
cp1           rpm=1.000e+00  power=-8.774e+07  heat=0.000e+00  nstages=1
              rat_cm=4.5347e+02  rat_crpm=5.7735e-02  rat_pr=6.0000  rat_eff=0.8800
              cm=1.0000e+00  crpm=1.0000e+00  pr=6.0000  eff=0.8800
ht1           heat=1.7278e+08
gt1           rpm=1.000e+00  power=1.277e+08
              rat_cm=1.3799e+02  rat_crpm=3.1623e-02  rat_pr=6.0000  rat_eff=0.8400
              cm=1.0000e+00  crpm=1.0000e+00  pr=6.0000  eff=0.8400
```

<u>Gas Turbine System with Parameter Study</u>

```
task: a  n=0  f=3.947171e+07
  x= 1.000000e+00
  c=-3.947171e+07
  h= 5.5825e+03 hs= 5.5825e+03 mu=0.00e+00 n=5.58e+03 s=5.58e+03 a=1.00e+00
```

```
task: a  n=1  f=1.560657e+00
  x= 7.571580e+01
  c=-1.560657e+00
  h= 8.7271e-12 hs= 8.7271e-12 mu=0.00e+00 n=8.73e-12 s=8.73e-12 a=1.00e+00

task: a  n=2  f=2.980232e-08
  x= 7.571580e+01
  c= 2.980232e-08

                        output of model gastype flows

  model          temp    pres   mass        enth         entr     dens      vol       qual
                  (K)   (atm)  (kg/s)      (J/kg)     (J/kg-K) (kg/m^3)  (m^3/s)

gas1             300.0    1.00 7.572e+01  2.499e+04  1.32e+04 8.19e-02 9.25e+02  1.00
cp1              544.4    6.00 7.572e+01  3.376e+06  1.39e+04 2.70e-01 2.80e+02  1.00
ht1              800.0    6.00 7.572e+01  7.023e+06  1.94e+04 1.84e-01 4.11e+02  1.00
gt1              527.8    1.00 7.572e+01  3.143e+06  2.09e+04 4.65e-02 1.63e+03  1.00

                      output of model parameters

gas1             id=THR-tH2 area=0.000e+00 dt=0.00e+00 dp=0.00e+00 dm=0.00e+00 dh=0.00e+00
cp1              rpm=1.000e+00  power=-2.537e+08  heat=0.000e+00  nstages=1
                 rat_cm=1.3114e+03  rat_crpm=5.7735e-02  rat_pr=6.0000  rat_eff=0.8800
                 cm=1.0000e+00  crpm=1.0000e+00  pr=6.0000  eff=0.8800
ht1              heat=2.7609e+08
gt1              rpm=1.000e+00  power=2.937e+08
                 rat_cm=3.5693e+02  rat_crpm=3.5355e-02  rat_pr=6.0000  rat_eff=0.8400
                 cm=1.0000e+00  crpm=1.0000e+00  pr=6.0000  eff=0.8400

task: a  n=0  f=3.847218e+07
  x= 1.000000e+00
  c=-3.847218e+07
  h= 6.3409e+02 hs= 6.3409e+02 mu=0.00e+00 n=6.34e+02 s=6.34e+02 a=1.00e+00

task: a  n=1  f=7.231588e-01
  x= 2.618110e+01
  c=-7.231588e-01
  h= 2.2404e-13 hs= 2.2404e-13 mu=0.00e+00 n=2.24e-13 s=2.24e-13 a=1.00e+00

task: a  n=2  f=2.980232e-08
  x= 2.618110e+01
  c= 2.980232e-08

                        output of model gastype flows

  model          temp    pres   mass        enth         entr     dens      vol       qual
                  (K)   (atm)  (kg/s)      (J/kg)     (J/kg-K) (kg/m^3)  (m^3/s)

gas1             300.0    1.00 2.618e+01  2.499e+04  1.32e+04 8.19e-02 3.20e+02  1.00
cp1              544.4    6.00 2.618e+01  3.376e+06  1.39e+04 2.70e-01 9.68e+01  1.00
ht1             1000.0    6.00 2.618e+01  9.976e+06  2.27e+04 1.47e-01 1.78e+02  1.00
gt1              666.4    1.00 2.618e+01  5.097e+06  2.42e+04 3.69e-02 7.10e+02  1.00

                      output of model parameters

gas1             id=THR-tH2 area=0.000e+00 dt=0.00e+00 dp=0.00e+00 dm=0.00e+00 dh=0.00e+00
```

```
cp1           rpm=1.000e+00  power=-8.774e+07  heat=0.000e+00  nstages=1
              rat_cm=4.5347e+02  rat_crpm=5.7735e-02  rat_pr=6.0000  rat_eff=0.8800
              cm=1.0000e+00  crpm=1.0000e+00  pr=6.0000  eff=0.8800
ht1           heat=1.7278e+08
gt1           rpm=1.000e+00  power=1.277e+08
              rat_cm=1.3799e+02  rat_crpm=3.1623e-02  rat_pr=6.0000  rat_eff=0.8400
              cm=1.0000e+00  crpm=1.0000e+00  pr=6.0000  eff=0.8400


task: a  n=0  f=3.746247e+07
  x= 1.000000e+00
  c=-3.746247e+07
  h= 2.1796e+02 hs= 2.1796e+02 mu=0.00e+00 n=2.18e+02 s=2.18e+02 a=1.00e+00

task: a  n=1  f=1.851436e-01
  x= 1.576336e+01
  c= 1.851436e-01
  h= 5.3235e-15 hs= 5.3235e-15 mu=0.00e+00 n=5.32e-15 s=5.32e-15 a=1.00e+00

task: a  n=2  f=0.000000e+00
  x= 1.576336e+01
  c= 0.000000e+00
                    output of model gastype flows

  model          temp   pres   mass        enth       entr      dens       vol       qual
                  (K)  (atm)  (kg/s)      (J/kg)   (J/kg-K)  (kg/m^3)  (m^3/s)

gas1            300.0   1.00 1.576e+01  2.499e+04  1.32e+04 8.19e-02 1.93e+02  1.00
cp1             544.4   6.00 1.576e+01  3.376e+06  1.39e+04 2.70e-01 5.83e+01  1.00
ht1            1200.0   6.00 1.576e+01  1.302e+07  2.55e+04 1.23e-01 1.28e+02  1.00
gt1             807.3   1.00 1.576e+01  7.127e+06  2.70e+04 3.04e-02 5.18e+02  1.00


                  output of model parameters

gas1          id=THR-tH2 area=0.000e+00 dt=0.00e+00 dp=0.00e+00 dm=0.00e+00 dh=0.00e+00
cp1           rpm=1.000e+00  power=-5.283e+07  heat=0.000e+00  nstages=1
              rat_cm=2.7303e+02  rat_crpm=5.7735e-02  rat_pr=6.0000  rat_eff=0.8800
              cm=1.0000e+00  crpm=1.0000e+00  pr=6.0000  eff=0.8800
ht1           heat=1.5195e+08
gt1           rpm=1.000e+00  power=9.283e+07
              rat_cm=9.1010e+01  rat_crpm=2.8868e-02  rat_pr=6.0000  rat_eff=0.8400
              cm=1.0000e+00  crpm=1.0000e+00  pr=6.0000  eff=0.8400


task: a  n=0  f=3.644363e+07
  x= 1.000000e+00
  c=-3.644363e+07
  h= 1.0501e+02 hs= 1.0501e+02 mu=0.00e+00 n=1.05e+02 s=1.05e+02 a=1.00e+00

task: a  n=1  f=5.054914e-01
  x= 1.124741e+01
  c= 5.054914e-01
  h= 2.0203e-14 hs= 2.0203e-14 mu=0.00e+00 n=2.02e-14 s=2.02e-14 a=1.00e+00

task: a  n=2  f=7.450581e-09
  x= 1.124741e+01
  c=-7.450581e-09
```

```
                         output of model gastype flows

  model          temp    pres   mass         enth        entr      dens       vol       qual
                  (K)   (atm)  (kg/s)        (J/kg)    (J/kg-K) (kg/m^3)  (m^3/s)

gas1             300.0   1.00 1.125e+01  2.499e+04  1.32e+04 8.19e-02 1.37e+02  1.00
cp1              544.4   6.00 1.125e+01  3.376e+06  1.39e+04 2.70e-01 4.16e+01  1.00
ht1             1400.0   6.00 1.125e+01  1.614e+07  2.79e+04 1.05e-01 1.07e+02  1.00
gt1              950.4   1.00 1.125e+01  9.232e+06  2.94e+04 2.58e-02 4.35e+02  1.00


                       output of model parameters

gas1          id=THR-tH2 area=0.000e+00 dt=0.00e+00 dp=0.00e+00 dm=0.00e+00 dh=0.00e+00
cp1           rpm=1.000e+00  power=-3.769e+07  heat=0.000e+00  nstages=1
              rat_cm=1.9481e+02  rat_crpm=5.7735e-02  rat_pr=6.0000  rat_eff=0.8800
              cm=1.0000e+00  crpm=1.0000e+00  pr=6.0000  eff=0.8800
ht1           heat=1.4356e+08
gt1           rpm=1.000e+00  power=7.769e+07
              rat_cm=7.0140e+01  rat_crpm=2.6726e-02  rat_pr=6.0000  rat_eff=0.8400
              cm=1.0000e+00  crpm=1.0000e+00  pr=6.0000  eff=0.8400
```

Space Propulsive System

```
thermodynamic data for HYDROGEN with flow id = THR-tH2
    pc=12.800000, tc=33.200000, tb=20.400000, molwt=2.016000

                         output of model gastype flows

  model          temp    pres   mass         enth        entr      dens       vol       qual
                  (K)   (atm)  (kg/s)        (J/kg)    (J/kg-K) (kg/m^3)  (m^3/s)

gas_h2            20.0   1.29 7.387e+00 -4.130e+06 -4.51e+04 7.74e+01 9.55e-02  0.00
pump_lp           20.6   7.90 7.387e+00 -4.117e+06 -4.49e+04 7.68e+01 9.62e-02  0.00
pump_hp           28.2 139.22 7.387e+00 -3.903e+06 -4.34e+04 7.55e+01 9.78e-02  1.00
hx_nz            583.3 139.22 7.387e+00  4.002e+06  1.93e+03 5.62e+00 1.31e+00  1.00
sp_2             583.3 139.22 5.171e+00  4.002e+06  1.93e+03 5.62e+00 9.20e-01  1.00
sp_1             583.3 139.22 3.620e+00  4.002e+06  1.93e+03 5.62e+00 6.44e-01  1.00
gt_lp            567.5  85.91 3.620e+00  3.744e+06  3.54e+03 3.63e+00 9.98e-01  1.00
mx_1.s           567.5  85.91 3.620e+00  3.744e+06  3.54e+03 3.63e+00 9.98e-01  1.00
sp_2.s           583.3 139.22 2.216e+00  4.002e+06  1.93e+03 5.62e+00 3.94e-01  1.00
mx_2.s           583.3 139.22 2.216e+00  4.002e+06  1.93e+03 5.62e+00 3.94e-01  1.00
sp_1.s           583.3 139.22 1.551e+00  4.002e+06  1.93e+03 5.62e+00 2.76e-01  1.00
gt_hp            526.0  85.91 1.551e+00  3.161e+06  2.47e+03 3.91e+00 3.97e-01  1.00
mx_1             555.1  85.91 5.171e+00  3.569e+06  3.23e+03 3.71e+00 1.40e+00  1.00
mx_2             564.3  85.91 7.387e+00  3.699e+06  3.46e+03 3.65e+00 2.03e+00  1.00
ht_reac         2930.0  85.91 7.387e+00  4.249e+07  2.95e+04 7.17e-01 1.03e+01  1.00
hx_nz.h         2492.0  85.91 7.387e+00  3.459e+07  2.66e+04 8.42e-01 8.78e+00  1.00
nz_1             842.2   0.10 7.387e+00  6.766e+06  3.60e+04 3.89e-03 1.90e+03  1.00


                       output of model parameters


gas_h2            id=THR-tH2    area=0.000e+00    dt=0.00e+00    dp=0.00e+00    dm=0.00e+00
dh=0.00e+00
pump_lp        mode=d  rpm=0.0000e+00  dp=6.6100e+00  eff=0.6700  power=-9.5452e+04
```

```
               rat_dp=6.6100e+00  rat_rpm=0.0000e+00  rat_m=7.3870e+00  rat_eff=0.6700
               inertia=0.0000e+00
pump_hp        mode=d  rpm=0.0000e+00  dp=1.3132e+02  eff=0.8100  power=-1.5810e+06
               rat_dp=1.3132e+02  rat_rpm=0.0000e+00  rat_m=7.3870e+00  rat_eff=0.8100
               inertia=0.0000e+00
hx_nz          type=count mode=d
               heat=5.8393e+07  lmtd=2404.778  pf_cold=0.00  pf_hot=0.00
               ufc=30.000  ufh=30.000  u=15.000  surfarea=1618.800
               denswall=7800.00  thickwall=1.000e-03  weight=6313.321
sp_2           sr=3.0000e-01
sp_1           sr=3.0000e-01
gt_lp          rpm=1.000e+00  power=9.330e+05
               rat_cm=6.2794e-01  rat_crpm=4.1404e-02  rat_pr=1.6205  rat_eff=0.2300
               cm=1.0000e+00  crpm=1.0000e+00  pr=1.6205  eff=0.2300
gt_hp          rpm=1.000e+00  power=1.304e+06
               rat_cm=2.6912e-01  rat_crpm=4.1404e-02  rat_pr=1.6205  rat_eff=0.7500
               cm=1.0000e+00  crpm=1.0000e+00  pr=1.6205  eff=0.7500
ht_reac        heat=2.8658e+08
nz_1           mode=d  eff=8.5000e-01  areain=4.3883e-02  area=3.2540e-01
               mach=3.8467e+00  thrust=6.1166e+04  impulse=8.4493e+02


                    output of model powers

  model          input         loss          prod          cons
                  (W)           (W)           (W)           (W)

pump_lp        0.0000e+00   0.0000e+00   0.0000e+00   9.5452e+04
pump_hp        0.0000e+00   0.0000e+00   0.0000e+00   1.5810e+06
gt_lp          0.0000e+00   0.0000e+00   9.3300e+05   0.0000e+00
gt_hp          0.0000e+00   0.0000e+00   1.3039e+06   0.0000e+00
ht_reac        2.8658e+08   0.0000e+00   0.0000e+00   0.0000e+00
totals         2.8658e+08   0.0000e+00   2.2369e+06   1.6765e+06

netprod        5.6040e+05
netinput       2.8658e+08
```

Space Propulsive System with Constraints

```
task: a  n=0  f=8.822084e+05
  x= 3.000000e-01  3.000000e-01
  c= 8.375455e+05 -2.771448e+05
  h= 3.9893e-01 hs= 3.9893e-01 mu=0.00e+00 n=2.11e-01 s=1.71e-01 a=9.48e-01

task: a  n=1  f=5.984499e+05
  x= 6.952333e-01  5.331519e-01
  c= 1.754595e+05 -5.721506e+05
  h= 3.4659e-02 hs= 3.4659e-02 mu=0.00e+00 n=2.37e-02 s=2.36e-02 a=9.99e-01

task: a  n=2  f=2.934375e+05
  x= 6.952333e-01  6.872562e-01
  c= 8.603293e+04 -2.805422e+05
  h= 8.3329e-03 hs= 8.3329e-03 mu=0.00e+00 n=2.20e-02 s=2.09e-02 a=9.96e-01

task: a  n=3  f=1.974849e-04
  x= 6.952333e-01  8.355123e-01
  c=-5.790050e-05  1.888063e-04
```

```
                     output of model gastype flows

  model          temp    pres   mass        enth         entr     dens     vol      qual
                 (K)    (atm)  (kg/s)      (J/kg)     (J/kg-K) (kg/m^3) (m^3/s)

gas_h2           20.0    1.29 7.387e+00 -4.130e+06 -4.51e+04 7.74e+01 9.55e-02  0.00
pump_lp          20.6    7.90 7.387e+00 -4.117e+06 -4.49e+04 7.68e+01 9.62e-02  0.00
pump_hp          28.2  139.22 7.387e+00 -3.903e+06 -4.34e+04 7.55e+01 9.78e-02  1.00
hx_nz           583.3  139.22 7.387e+00  4.002e+06  1.93e+03 5.62e+00 1.31e+00  1.00
sp_2            583.3  139.22 2.251e+00  4.002e+06  1.93e+03 5.62e+00 4.00e-01  1.00
sp_1            583.3  139.22 3.703e-01  4.002e+06  1.93e+03 5.62e+00 6.59e-02  1.00
gt_lp           567.5   85.91 3.703e-01  3.744e+06  3.54e+03 3.63e+00 1.02e-01  1.00
mx_1.s          567.5   85.91 3.703e-01  3.744e+06  3.54e+03 3.63e+00 1.02e-01  1.00
sp_2.s          583.3  139.22 5.136e+00  4.002e+06  1.93e+03 5.62e+00 9.13e-01  1.00
mx_2.s          583.3  139.22 5.136e+00  4.002e+06  1.93e+03 5.62e+00 9.13e-01  1.00
sp_1.s          583.3  139.22 1.881e+00  4.002e+06  1.93e+03 5.62e+00 3.34e-01  1.00
gt_hp           526.0   85.91 1.881e+00  3.161e+06  2.47e+03 3.91e+00 4.82e-01  1.00
mx_1            532.8   85.91 2.251e+00  3.257e+06  2.65e+03 3.86e+00 5.84e-01  1.00
mx_2            569.7   85.91 7.387e+00  3.775e+06  3.59e+03 3.61e+00 2.04e+00  1.00
ht_reac        2930.0   85.91 7.387e+00  4.249e+07  2.95e+04 7.17e-01 1.03e+01  1.00
hx_nz.h        2492.0   85.91 7.387e+00  3.459e+07  2.66e+04 8.42e-01 8.78e+00  1.00
nz_1            842.2    0.10 7.387e+00  6.766e+06  3.60e+04 3.89e-03 1.90e+03  1.00


                     output of model parameters

gas_h2       id=THR-tH2 area=0.000e+00 dt=0.00e+00  dp=0.00e+00 dm=0.00e+00 dh=0.00e+00
pump_lp       mode=d  rpm=0.0000e+00  dp=6.6100e+00  eff=0.6700  power=-9.5452e+04
              rat_dp=6.6100e+00  rat_rpm=0.0000e+00  rat_m=7.3870e+00  rat_eff=0.6700
              inertia=0.0000e+00
pump_hp       mode=d  rpm=0.0000e+00  dp=1.3132e+02  eff=0.8100  power=-1.5810e+06
              rat_dp=1.3132e+02  rat_rpm=0.0000e+00  rat_m=7.3870e+00  rat_eff=0.8100
              inertia=0.0000e+00
hx_nz         type=count mode=d
              heat=5.8393e+07  lmtd=2404.778  pf_cold=0.00  pf_hot=0.00
              ufc=30.000  ufh=30.000  u=15.000  surfarea=1618.800
              denswall=7800.00  thickwall=1.000e-03  weight=6313.321
sp_2          sr=6.9523e-01
sp_1          sr=8.3551e-01
gt_lp         rpm=1.000e+00  power=9.545e+04
              rat_cm=6.4243e-02  rat_crpm=4.1404e-02  rat_pr=1.6205  rat_eff=0.2300
              cm=1.0000e+00  crpm=1.0000e+00  pr=1.6205  eff=0.2300
gt_hp         rpm=1.000e+00  power=1.581e+06
              rat_cm=3.2632e-01  rat_crpm=4.1404e-02  rat_pr=1.6205  rat_eff=0.7500
              cm=1.0000e+00  crpm=1.0000e+00  pr=1.6205  eff=0.7500
ht_reac       heat=2.8602e+08
nz_1          mode=d  eff=8.5000e-01  areain=4.3883e-02  area=3.2540e-01
              mach=3.8467e+00  thrust=6.1166e+04  impulse=8.4493e+02


                     output of model powers

  model          input          loss          prod          cons
                  (W)            (W)           (W)           (W)

pump_lp       0.0000e+00    0.0000e+00    0.0000e+00    9.5452e+04
pump_hp       0.0000e+00    0.0000e+00    0.0000e+00    1.5810e+06
gt_lp         0.0000e+00    0.0000e+00    9.5452e+04    0.0000e+00
gt_hp         0.0000e+00    0.0000e+00    1.5810e+06    0.0000e+00
ht_reac       2.8602e+08    0.0000e+00    0.0000e+00    0.0000e+00
```

```
totals          2.8602e+08    0.0000e+00    1.6765e+06    1.6765e+06

netprod         1.3091e-04
netinput        2.8602e+08
```

Coal-Fired Power Plant

```
task: a  n=0  f=1.106367e+05
  x=-5.000000e-01
  c=-1.106367e+05
  h= 2.1462e-02 hs= 2.1462e-02 mu=0.00e+00 n=2.15e-02 s=2.15e-02 a=1.00e+00

task: a  n=1  f=1.015648e-03
  x=-3.535023e-01
  c= 1.015648e-03
  h= 1.8086e-18 hs= 1.8086e-18 mu=0.00e+00 n=1.81e-18 s=1.81e-18 a=1.00e+00

task: a  n=2  f=0.000000e+00
  x=-3.535023e-01
  c= 0.000000e+00

task: b  n=0  f=1.082036e+01
  x= 1.500000e+01
  c= 1.082036e+01
  h= 7.4673e-03 hs= 7.4673e-03 mu=0.00e+00 n=7.47e-03 s=7.47e-03 a=1.00e+00

task: b  n=1  f=5.665290e-02
  x= 1.491359e+01
  c=-5.665290e-02
  h= 2.0470e-07 hs= 2.0470e-07 mu=0.00e+00 n=2.03e-07 s=2.03e-07 a=1.00e+00

task: b  n=2  f=3.433942e-04
  x= 1.491404e+01
  c=-3.433942e-04

                     output of model parameters

gas_air        id=GAS  area=0.000e+00  dt=0.00e+00  dp=0.00e+00  dm=0.00e+00  dh=0.00e+00
cb_gas         stoich=1.3558e+00  mass=1.4914e+01  lhv=3.2140e+07
gas_wat        id=STM  area=0.000e+00  dt=2.09e-10  dp=0.00e+00  dm=0.00e+00  dh=0.00e+00
pump_fw        mode=d  rpm=0.0000e+00  dp=1.7993e+02  eff=0.8500  power=-2.6984e+06
               rat_dp=1.7993e+02  rat_rpm=0.0000e+00  rat_m=1.2500e+02  rat_eff=0.8500
               inertia=0.0000e+00
hx_boil        type=count mode=d
               heat=2.2789e+08  lmtd=1225.646  pf_cold=0.00  pf_hot=0.00
               ufc=30.000  ufh=30.000  u=15.000  surfarea=12395.415
               denswall=7800.00  thickwall=1.000e-03  weight=48342.120
hx_sh          type=count mode=d
               heat=1.1031e+08  lmtd=599.386  pf_cold=0.00  pf_hot=0.00
               ufc=30.000  ufh=30.000  u=15.000  surfarea=12268.939
               denswall=7800.00  thickwall=1.000e-03  weight=47848.862
hx_rh          type=count mode=d
               heat=5.0650e+07  lmtd=321.381  pf_cold=0.00  pf_hot=0.00
               ufc=30.000  ufh=30.000  u=15.000  surfarea=10506.795
               denswall=7800.00  thickwall=1.000e-03  weight=40976.501
hx_air         type=count mode=d
               heat=6.3864e+07  lmtd=372.380  pf_cold=0.00  pf_hot=0.00
               ufc=30.000  ufh=30.000  u=15.000  surfarea=11433.548
```

```
                 denswall=7800.00  thickwall=1.000e-03  weight=44590.838
st_hp            rpm=1.000e+00  power=3.835e+07
                 rat_cm=1.9776e+01  rat_crpm=3.5115e-02  rat_pr=3.6000  rat_eff=0.8400
                 cm=1.0000e+00  crpm=1.0000e+00  pr=3.6000  eff=0.8400
st_lp            rpm=1.000e+00  power=1.284e+08
                 rat_cm=6.4076e+01  rat_crpm=3.5115e-02  rat_pr=757.5758  rat_eff=0.8600
                 cm=1.0000e+00  crpm=1.0000e+00  pr=757.5758  eff=0.8600
fh_hp            cascade=0  subcool=1.0000e+01
                 htemp=6.2478e+02 5.3788e+02 5.3788e+02 5.2788e+02
                 ctemp=3.8353e+02 3.8471e+02 4.2345e+02 4.2996e+02
                 q=3.4713e+06 2.0443e+07 6.1983e+05
fh_lp            cascade=1  subcool=1.0000e+01
                 htemp=3.1132e+02 3.1132e+02 3.1132e+02 3.0132e+02
                 ctemp=3.1072e+02 3.1264e+02 3.8353e+02 3.8353e+02
                 q=0.0000e+00 3.6828e+07 9.9219e+05
ht_cond          heat=-2.2475e+08


                           output of model gastype flows

  model            temp    pres   mass         enth          entr      dens      vol       qual
                   (K)     (atm)  (kg/s)       (J/kg)      (J/kg-K)  (kg/m^3)  (m^3/s)

gas_wat           597.4 180.00 5.000e+02 -1.450e+07  6.97e+03 6.77e+02 7.38e-01 -0.35
hx_boil           631.1 180.00 5.000e+02 -1.404e+07  7.70e+03 3.12e+02 1.60e+00  0.25
sd_1              631.1 180.00 3.750e+02 -1.423e+07  7.40e+03 5.39e+02 6.95e-01  0.00
mx_fw.s           631.1 180.00 3.750e+02 -1.423e+07  7.40e+03 5.39e+02 6.95e-01  0.00
sd_1.s            631.1 180.00 1.250e+02 -1.348e+07  8.60e+03 1.38e+02 9.06e-01  1.00
hx_sh             811.0 180.00 1.250e+02 -1.259e+07  9.87e+03 5.57e+01 2.25e+00  2.17
st_hp             624.8  50.00 1.125e+02 -1.290e+07  9.96e+03 1.95e+01 5.78e+00  1.17
hx_rh             811.0  50.00 1.125e+02 -1.245e+07  1.06e+04 1.40e+01 8.04e+00  1.45
st_lp             311.3   0.07 1.013e+02 -1.359e+07  1.12e+04 5.07e-02 2.00e+03  0.92
ht_cond           311.3   0.07 1.013e+02 -1.581e+07  4.06e+03 9.93e+02 1.02e-01  0.00
mx_ext.s          311.3   0.07 1.013e+02 -1.581e+07  4.06e+03 9.93e+02 1.02e-01  0.00
st_hp.s           624.8  50.00 1.250e+01 -1.290e+07  9.96e+03 1.95e+01 6.42e-01  1.17
fh_hp.h           527.9  50.00 1.250e+01 -1.486e+07  6.35e+03 7.93e+02 1.58e-02  0.00
st_lp.s           311.3   0.07 1.125e+01 -1.359e+07  1.12e+04 5.07e-02 2.22e+02  0.92
fh_lp.h           301.3   0.07 2.375e+01 -1.585e+07  3.93e+03 9.96e+02 2.38e-02  0.00
mx_ext            309.4   0.07 1.250e+02 -1.582e+07  4.04e+03 9.94e+02 1.26e-01 -0.00
pump_fw           310.7 180.00 1.250e+02 -1.580e+07  4.05e+03 1.00e+03 1.25e-01 -2.07
fh_lp             383.5 180.00 1.250e+02 -1.550e+07  4.92e+03 9.59e+02 1.30e-01 -1.67
fh_hp             430.0 180.00 1.250e+02 -1.530e+07  5.41e+03 9.21e+02 1.36e-01 -1.41
mx_fw             597.4 180.00 5.000e+02 -1.450e+07  6.97e+03 6.77e+02 7.38e-01 -0.35
gas_wat.cyc       597.4 180.00 5.000e+02 -1.450e+07  6.97e+03 6.77e+02 7.38e-01 -0.35
gas_air           300.0   1.00 2.067e+02  1.861e+03  6.89e+03 1.17e+00 1.76e+02  1.00
hx_air            600.0   1.00 2.067e+02  3.108e+05  7.60e+03 5.86e-01 3.53e+02  1.00
cb_gas           2219.3   1.00 2.216e+02  2.568e+05  9.21e+03 1.62e-01 1.36e+03  1.00
hx_boil.h        1520.5   1.00 2.216e+02 -7.715e+05  8.65e+03 2.38e-01 9.31e+02  1.00
hx_sh.h          1132.5   1.00 2.216e+02 -1.269e+06  8.28e+03 3.20e-01 6.94e+02  1.00
hx_rh.h           946.1   1.00 2.216e+02 -1.498e+06  8.06e+03 3.82e-01 5.79e+02  1.00
hx_air.h          700.0   1.00 2.216e+02 -1.786e+06  7.70e+03 5.17e-01 4.29e+02  1.00


     output of species molar flow rates(gmol/s) and mole fractions(%)

gas_wat            H2O=0.00e+00  H2Oc= 27754
hx_boil            H2O=6938.5  H2Oc= 20816
sd_1               H2O=0.00e+00  H2Oc= 20816
mx_fw.s            H2O=0.00e+00  H2Oc= 20816
sd_1.s             H2O=6938.5  H2Oc=0.00e+00
```

```
hx_sh           H2O=6938.5  H2Oc=0.00e+00
st_hp           H2O=6244.7  H2Oc=0.00e+00
hx_rh           H2O=6244.7  H2Oc=0.00e+00
st_lp           H2O=5174.2  H2Oc=446.02
ht_cond         H2O=0.00e+00  H2Oc=5620.2
mx_ext.s        H2O=0.00e+00  H2Oc=5620.2
st_hp.s         H2O=693.85  H2Oc=0.00e+00
fh_hp.h         H2O=0.00e+00  H2Oc=693.85
st_lp.s         H2O=574.91  H2Oc=49.558
fh_lp.h         H2O=0.00e+00  H2Oc=1318.3
mx_ext          H2O=0.00e+00  H2Oc=6938.5
pump_fw         H2O=0.00e+00  H2Oc=6938.5
fh_lp           H2O=0.00e+00  H2Oc=6938.5
fh_hp           H2O=0.00e+00  H2Oc=6938.5
mx_fw           H2O=0.00e+00  H2Oc= 27754
gas_wat.cyc     H2O=0.00e+00  H2Oc= 27754
gas_air         O2=1504.5 21  N2=5659.9 79  total=7164.5
hx_air          O2=1504.5 21  N2=5659.9 79  NO=0.0002 0   total=7164.5
cb_gas        CO=34.468 0  CO2=930.19 12 H=1.1717 0  H2=3.1276 0  H2O=444.13 6  O=6.2708 0
             O2=378.71 5  OH=23.638 0  N2=5617.5 75 NO=49.861 1  SO2=3.385 0 total=7492.5
hx_boil.h      CO=0.0339 0  CO2=964.62 13 H=0.0002 0  H2=0.0061 0  H2O=459.45 6 O=0.0111 0
             O2=390.84 5  OH=0.4151 0  N2=5639.8 76 NO=5.3282 0  SO2=3.385 0 total=7463.9
hx_sh.h       CO2=964.66 13 H2O=459.66 6 O2=393.36 5 OH=0.0050 0 N2=5642.2 76 NO=0.4603 0
                SO2=3.3850 0  total=7463.8
hx_rh.h       CO2=964.66 13 H2O=459.66 6 O2=393.56 5 OH=0.0002 0 N2=5642.4 76 NO=0.0694 0
                SO2=3.3850 0  total=7463.8
hx_air.h      CO2=964.66 13 H2O=459.66 6 O2=393.59 5 N2=5642.5 76 NO=0.0012 0 SO2=3.385 0
                total=7463.8


                     output of model powers

  model          input         loss          prod          cons
                  (W)           (W)           (W)           (W)

st_hp          0.0000e+00    0.0000e+00    3.8353e+07    0.0000e+00
st_lp          0.0000e+00    0.0000e+00    1.2844e+08    0.0000e+00
ht_cond        0.0000e+00    2.2475e+08    0.0000e+00    0.0000e+00
pump_fw        0.0000e+00    0.0000e+00    0.0000e+00    2.6984e+06
gas_wat        0.0000e+00    0.0000e+00    0.0000e+00    0.0000e+00
cb_gas         4.7934e+08    0.0000e+00    0.0000e+00    0.0000e+00
totals         4.7934e+08    2.2475e+08    1.6679e+08    2.6984e+06

netprod        1.6409e+08
netinput       2.5459e+08
```

<u>PEM Fuel Cell System</u>

```
task: task_1  n=0  f=4.045676e+00
  x= 6.000000e-01  1.100000e+00  3.250000e+02
  c= 5.973979e-03 -4.045672e+00 -4.665418e-04
  h= 4.9267e-01 hs= 4.9267e-01 mu=0.00e+00 n=4.63e-01 s=3.64e-01 a=9.61e-01

task: task_1  n=1  f=1.411664e+00
  x= 6.466005e-01  8.766314e-01  3.256411e+02
  c= 5.202500e-04  1.411663e+00  1.338197e-05
  h= 1.1273e-02 hs= 1.1273e-02 mu=0.00e+00 n=9.60e-03 s=6.00e-03 a=8.99e-01
```

```
task: task_1  n=2  f=9.389739e-01
  x= 6.480362e-01  9.720523e-01  3.256188e+02
  c=-3.984894e-04 -9.389738e-01 -3.544908e-06
  h= 4.8663e-03 hs= 4.8663e-03 mu=0.00e+00 n=1.55e-03 s=1.33e-03 a=9.53e-01

task: task_1  n=3  f=3.728309e-02
  x= 6.472375e-01  9.329366e-01  3.256235e+02
  c=-8.088804e-06 -3.728309e-02  1.990278e-08
  h= 7.6277e-06 hs= 7.6277e-06 mu=0.00e+00 n=2.73e-06 s=2.21e-06 a=9.39e-01

task: task_1  n=4  f=2.231653e-03
  x= 6.472580e-01  9.312844e-01  3.256235e+02
  c= 5.283398e-07  2.231653e-03 -3.682667e-09
  h= 2.7353e-08 hs= 2.7353e-08 mu=0.00e+00 n=8.87e-09 s=7.26e-09 a=9.40e-01

task: task_1  n=5  f=1.826387e-05
  x= 6.472572e-01  9.313784e-01  3.256235e+02
  c=-8.144117e-09 -1.826387e-05 -2.691958e-11

                    output of model parameters

air          id=GAS  area=0.000e+00  dt=0.00e+00  dp=0.00e+00  dm=0.00e+00  dh=0.00e+00
              humid=0.500  pvap=0.035
fuel         id=THR-CH4O area=0.000e+00 dt=0.00e+00 dp=0.00e+00 dm=0.00e+00 dh=0.00e+00
air_cond     id=GAS  area=0.000e+00  dt=0.00e+00  dp=0.00e+00  dm=0.00e+00  dh=0.00e+00
              humid=0.500  pvap=0.035
air_rej      id=GAS  area=0.000e+00  dt=0.00e+00  dp=0.00e+00  dm=0.00e+00  dh=0.00e+00
              humid=0.500  pvap=0.035
air_int      id=GAS  area=0.000e+00  dt=0.00e+00  dp=0.00e+00  dm=0.00e+00  dh=0.00e+00
              humid=0.500  pvap=0.035
pump_fuel     mode=d  rpm=0.0000e+00  dp=2.0000e+00  eff=0.7500  power=-3.8913e+00
              rat_dp=2.0000e+00  rat_rpm=0.0000e+00  rat_m=1.0574e-02  rat_eff=0.7500
              inertia=0.0000e+00
pump_water    mode=d  rpm=0.0000e+00  dp=1.0000e+00  eff=0.7500  power=-1.2750e+02
              rat_dp=1.0000e+00  rat_rpm=0.0000e+00  rat_m=9.3138e-01  rat_eff=0.7500
              inertia=0.0000e+00
wat          id=STM  area=0.000e+00  dt=0.00e+00  dp=0.00e+00  dm=0.00e+00  dh=1.14e-01
water_tank    id=STM area=0.000e+00 dt=-2.07e-02 dp=0.00e+00 dm=-2.69e-11 dh=-8.63e+01
h2o          id=STM  area=0.000e+00  dt=0.00e+00  dp=0.00e+00  dm=0.00e+00  dh=0.00e+00
hx_preh       type=count mode=d
              heat=1.8212e+03  lmtd=18.902  pf_cold=0.00  pf_hot=0.00
              ufc=30.000  ufh=50.000  u=18.750  surfarea=5.139
              denswall=7800.00  thickwall=1.000e-04  weight=2.004
hx_rej        type=count mode=d
              heat=8.5127e+04  lmtd=27.722  pf_cold=0.00  pf_hot=0.00
              ufc=30.000  ufh=50.000  u=18.750  surfarea=163.773
              denswall=7800.00  thickwall=1.000e-04  weight=63.872
hx_cool       type=count mode=d
              heat=1.1538e+04  lmtd=50.570  pf_cold=0.00  pf_hot=0.00
              ufc=30.000  ufh=50.000  u=18.750  surfarea=12.169
              denswall=7800.00  thickwall=1.000e-04  weight=4.746
sp_air        sr=2.6660e-01
sp_prox       sr=9.8277e-01
sp_wat        sr=7.6288e-03
sp_shif       sr=2.4961e-02
sp_fuel       sr=1.0000e-01
sp_anode      sr=0.0000e+00
sp_h2o        sr=-1.0000e+00
              ssr[H2Oc]=1.000e+00
```

```
cond_1          h2oin=9.367e-04  h2oout=7.477e-04  h2ocond=1.891e-04
                heat=1.383e+04  pvap=1.376e-01  ph2o=1.376e-01
                lmtd=3.408e+01  u=3.000e+01  area=1.353e+01
                dens=7800.00  thick=5.000e-04  weight=26.382
gt_1            rpm=1.000e+00  power=1.606e+04
                rat_cm=1.1016e+00  rat_crpm=4.7072e-02  rat_pr=3.0000  rat_eff=0.8000
                cm=1.0000e+00  crpm=1.0000e+00  pr=3.0000  eff=0.8000
cp_air          rpm=1.000e+00  power=-1.847e+04  heat=-8.435e+03  nstages=2
                lmtd=1.405e+01  dt=2.001e+01  u=5.000e+00  area=30.000
                dens=2700.000  thick=1.000e-03  weight=18.971
                rat_cm=2.4522e+00  rat_crpm=5.7735e-02  rat_pr=1.7321  rat_eff=0.8000
                cm=1.0000e+00  crpm=1.0000e+00  pr=1.7321  eff=0.8000
fan_cond        rpm=1.000e+00  power=-5.511e+02  heat=0.000e+00  nstages=1
                rat_cm=1.7516e+01  rat_crpm=5.7735e-02  rat_pr=1.0050  rat_eff=0.8000
                cm=1.0000e+00  crpm=1.0000e+00  pr=1.0050  eff=0.8000
fan_int         rpm=1.000e+00  power=-5.511e+02  heat=0.000e+00  nstages=1
                rat_cm=1.7516e+01  rat_crpm=5.7735e-02  rat_pr=1.0050  rat_eff=0.8000
                cm=1.0000e+00  crpm=1.0000e+00  pr=1.0050  eff=0.8000
fan_rej         rpm=1.000e+00  power=-2.756e+03  heat=0.000e+00  nstages=1
                rat_cm=8.7580e+01  rat_crpm=5.7735e-02  rat_pr=1.0050  rat_eff=0.8000
                cm=1.0000e+00  crpm=1.0000e+00  pr=1.0050  eff=0.8000
cp_anode        rpm=1.000e+00  power=-9.660e+02  heat=0.000e+00  nstages=1
                rat_cm=1.6230e-01  rat_crpm=5.3225e-02  rat_pr=1.5000  rat_eff=0.8000
                cm=1.0000e+00  crpm=1.0000e+00  pr=1.5000  eff=0.8000
form            heat=3.706e+04  lmtd=3.786e+02  tmix=3.645e+02  ua=9.788e+01  type=paral
pefc            mode=d  celltemp=353.00  dtcell=0.000e+00
                h2reac=7.352e-04  h2odiff=-1.774e-05
                h2oain=1.261e-04  h2oaout=1.439e-04  increase=1.774e-05
                h2ocin=6.319e-05  h2ocout=7.807e-04  increase=7.175e-04
                pvap=0.465  ph2oa=0.464  ph2oc=0.465
                dh=-2.424e+08  dg=-2.283e+08  hdel=-1.886e+05
                option=ts  curden=0.575  voltideal=1.183  voltact=0.715
                fuelutil=0.850  o2util=0.500  loadr=5.038e-02
                lmtd=13.157  u=1.342e+02  area=2.4675e+01  w_p=1.600  weight=162.254
                power=1.014e+05  heatgen=7.679e+04  heat=8.714e+04
                effideal=0.942  effvolt=0.604  effact=0.569
```

```
                            output of model gastype flows

  model            temp     pres   mass        enth        entr      dens       vol       qual
                    (K)     (atm)  (kg/s)      (J/kg)    (J/kg-K)  (kg/m^3)  (m^3/s)

air                300.0    1.00 1.416e-01 -1.479e+05  6.96e+03 1.16e+00 1.22e-01  1.00
cp_air             369.3    3.00 1.416e-01 -7.700e+04  6.85e+03 2.84e+00 4.99e-02  1.00
sp_air             369.3    3.00 1.038e-01 -7.700e+04  6.85e+03 2.84e+00 3.66e-02  1.00
sp_prox            369.3    3.00 1.789e-03 -7.700e+04  6.85e+03 2.84e+00 6.31e-04  1.00
mx_prox.s          369.3    3.00 1.789e-03 -7.700e+04  6.85e+03 2.84e+00 6.31e-04  1.00
wat                393.8    2.00 7.134e-03 -1.404e+07  8.67e+03 1.76e+00 4.04e-03  0.65
sp_shif            393.8    2.00 6.956e-03 -1.404e+07  8.67e+03 1.76e+00 3.94e-03  0.65
form.s             393.8    2.00 6.956e-03 -1.404e+07  8.67e+03 1.76e+00 3.94e-03  0.65
fuel               300.0    1.00 1.057e-02 -7.549e+06  1.43e+04 7.34e+02 1.44e-05  0.00
pump_fuel          300.1    3.00 1.057e-02 -7.549e+06  1.43e+04 7.34e+02 1.44e-05  0.00
hx_preh            343.0    3.00 1.057e-02 -7.377e+06  1.48e+04 6.84e+02 1.55e-05  0.00
sp_fuel            343.0    3.00 9.516e-03 -7.377e+06  1.48e+04 6.84e+02 1.39e-05  0.00
form               473.1    2.00 1.647e-02 -7.941e+06  1.32e+04 6.64e-01 2.48e-02  0.00
mx_shif.s          473.1    2.00 1.647e-02 -7.941e+06  1.32e+04 6.64e-01 2.48e-02  0.00
sp_shif.s          393.8    2.00 1.781e-04 -1.404e+07  8.67e+03 1.76e+00 1.01e-04  0.65
mx_shif            470.4    2.00 1.665e-02 -8.006e+06  1.32e+04 6.70e-01 2.48e-02  1.00
mx_prox            582.6    2.00 1.844e-02 -7.237e+06  1.33e+04 5.77e-01 3.19e-02  1.00
```

```
hx_cool.h         353.0   2.00 1.844e-02 -7.863e+06  1.19e+04 9.53e-01 1.94e-02  1.00
sp_prox.s         369.3   3.00 1.020e-01 -7.700e+04  6.85e+03 2.84e+00 3.60e-02  1.00
pefc              353.0   3.00 1.032e-01 -1.850e+06  7.14e+03 2.89e+00 3.58e-02  1.00
sp_h2o            353.0   3.00 9.948e-02 -1.334e+06  7.24e+03 2.78e+00 3.58e-02  1.00
mx_cath.s         353.0   3.00 9.948e-02 -1.334e+06  7.24e+03 2.78e+00 3.58e-02  1.00
sp_air.s          369.3   3.00 3.775e-02 -7.700e+04  6.85e+03 2.84e+00 1.33e-02  1.00
mx_burn.s         369.3   3.00 3.775e-02 -7.700e+04  6.85e+03 2.84e+00 1.33e-02  1.00
sp_fuel.s         343.0   3.00 1.057e-03 -7.377e+06  1.48e+04 6.84e+02 1.55e-06  0.00
mx_fuel.s         343.0   3.00 1.057e-03 -7.377e+06  1.48e+04 6.84e+02 1.55e-06  0.00
pefc.a            353.0   2.00 1.728e-02 -8.707e+06  7.13e+03 1.93e+00 8.97e-03  1.00
cp_anode          396.5   3.00 1.728e-02 -8.651e+06  7.16e+03 2.57e+00 6.71e-03  1.00
sp_anode          396.5   3.00 1.728e-02 -8.651e+06  7.16e+03 2.57e+00 6.71e-03  1.00
mx_burn           871.0   3.00 5.502e-02 -2.769e+06  7.90e+03 1.23e+00 4.46e-02  1.00
mx_fuel          1140.7   3.00 5.608e-02 -2.856e+06  8.31e+03 9.36e-01 5.99e-02  1.00
form.h            620.7   3.00 5.608e-02 -3.517e+06  7.55e+03 1.72e+00 3.26e-02  1.00
mx_cath           451.3   3.00 1.556e-01 -2.121e+06  7.41e+03 2.24e+00 6.94e-02  1.00
mx_anode.s        451.3   3.00 1.556e-01 -2.121e+06  7.41e+03 2.24e+00 6.94e-02  1.00
sp_anode.s        396.5   3.00 0.000e+00 -8.651e+06  7.16e+03 2.57e+00 0.00e+00  1.00
mx_anode          451.3   3.00 1.556e-01 -2.121e+06  7.41e+03 2.24e+00 6.94e-02  1.00
gt_1              358.9   1.00 1.556e-01 -2.224e+06  7.49e+03 9.39e-01 1.66e-01  1.00
cond_1            325.6   1.00 1.522e-01 -2.012e+06  7.29e+03 1.05e+00 1.45e-01  1.00
sp_h2o.s          353.0   3.00 3.727e-03 -1.564e+07  4.59e+03 1.00e+03 3.73e-06  1.00
h2o               353.0   3.00 3.727e-03 -1.564e+07  4.59e+03 9.72e+02 3.84e-06  0.00
mx_h2o.s          353.0   3.00 3.727e-03 -1.564e+07  4.59e+03 9.72e+02 3.84e-06  0.00
water_tank        325.6   1.00 9.314e-01 -1.575e+07  4.25e+03 9.87e+02 9.44e-04 -0.09
pump_water        325.6   2.00 9.314e-01 -1.575e+07  4.25e+03 9.87e+02 9.44e-04 -0.13
pefc.cool         348.0   2.00 9.314e-01 -1.566e+07  4.53e+03 9.75e+02 9.55e-04 -0.09
mx_h2o            348.0   2.00 9.351e-01 -1.566e+07  4.53e+03 9.75e+02 9.59e-04 -0.09
sp_wat            348.0   2.00 9.280e-01 -1.566e+07  4.53e+03 9.75e+02 9.52e-04 -0.09
hx_preh.h         347.6   2.00 9.280e-01 -1.566e+07  4.52e+03 9.75e+02 9.52e-04 -0.09
hx_rej.h          325.6   2.00 9.280e-01 -1.575e+07  4.25e+03 9.87e+02 9.40e-04 -0.13
mx_cond.s         325.6   2.00 9.280e-01 -1.575e+07  4.25e+03 9.87e+02 9.40e-04 -0.13
cond_1.s          325.6   1.00 3.406e-03 -1.575e+07  4.25e+03 9.87e+02 3.45e-06  0.00
mx_cond           325.6   1.00 9.314e-01 -1.575e+07  4.25e+03 9.87e+02 9.44e-04 -0.09
water_tank.cyc    325.6   1.00 9.314e-01 -1.575e+07  4.25e+03 9.87e+02 9.44e-04 -0.09
sp_wat.s          348.0   2.00 7.134e-03 -1.566e+07  4.53e+03 9.75e+02 7.32e-06 -0.09
hx_cool           393.8   2.00 7.134e-03 -1.404e+07  8.67e+03 1.76e+00 4.04e-03  0.65
wat.cyc           393.8   2.00 7.134e-03 -1.404e+07  8.67e+03 1.76e+00 4.04e-03  0.65
air_int           300.0   1.00 1.011e+00 -1.479e+05  6.96e+03 1.16e+00 8.69e-01  1.00
fan_int           300.5   1.00 1.011e+00 -1.473e+05  6.96e+03 1.17e+00 8.66e-01  1.00
cp_air.cool       308.7   1.00 1.011e+00 -1.390e+05  6.98e+03 1.14e+00 8.89e-01  1.00
air_cond          300.0   1.00 1.011e+00 -1.479e+05  6.96e+03 1.16e+00 8.69e-01  1.00
fan_cond          300.5   1.00 1.011e+00 -1.473e+05  6.96e+03 1.17e+00 8.66e-01  1.00
cond_1.cool       313.9   1.00 1.011e+00 -1.336e+05  7.00e+03 1.12e+00 9.05e-01  1.00
air_rej           300.0   1.00 5.056e+00 -1.479e+05  6.96e+03 1.16e+00 4.34e+00  1.00
fan_rej           300.5   1.00 5.056e+00 -1.473e+05  6.96e+03 1.17e+00 4.33e+00  1.00
hx_rej            317.0   1.00 5.056e+00 -1.305e+05  7.01e+03 1.11e+00 4.57e+00  1.00


     output of species molar flow rates(gmol/s) and mole fractions(%)

air               H2O=0.0877 2    O2=1.0191 21   N2=3.8336 78   total=4.9403
cp_air            H2O=0.0877 2    O2=1.0191 21   N2=3.8336 78   total=4.9403
sp_air            H2O=0.0643 2    O2=0.7474 21   N2=2.8115 78   total=3.6232
sp_prox           H2O=0.0011 2    O2=0.0129 21   N2=0.0484 78   total=0.0624
mx_prox.s         H2O=0.0011 2    O2=0.0129 21   N2=0.0484 78   total=0.0624
wat               H2O=0.2563  H2Oc=0.1397
sp_shif           H2O=0.2499  H2Oc=0.1362
form.s            H2O=0.2499  H2Oc=0.1362
fuel              CH4O=0.3300
```

```
pump_fuel       CH4O=0.3300
hx_preh         CH4O=0.3300
sp_fuel         CH4O=0.2970
form            CO=0.0106 1   CO2=0.2864 22  H2=0.8804 69  H2O=0.0997 8   total=1.2771
mx_shif.s       CO=0.0106 1   CO2=0.2864 22  H2=0.8804 69  H2O=0.0997 8   total=1.2771
sp_shif.s       H2O=0.0064  H2Oc=0.0035
mx_shif         CO=0.0093 1   CO2=0.2877 22  H2=0.8817 69  H2O=0.1083 8   total=1.2870
mx_prox         CO=0.0369 3   CO2=0.2601 19  H2=0.8284 62  H2O=0.1627 12  N2=0.0484 4
                total=1.3365
hx_cool.h       CO=0.0003 0   CO2=0.2967 22  H2=0.8650 65  H2O=0.1261 9   N2=0.0484 4
                total=1.3365
sp_prox.s       H2O=0.0632 2   O2=0.7345 21  N2=2.7631 78  total=3.5608
pefc            H2O=0.5738 15  O2=0.3669 9   N2=2.7631 71  H2Oc=0.2069 5   total=3.9107
sp_h2o          H2O=0.5738 15  O2=0.3669 10  N2=2.7631 75  total=3.7038
mx_cath.s       H2O=0.5738 15  O2=0.3669 10  N2=2.7631 75  total=3.7038
sp_air.s        H2O=0.0234 2   O2=0.2717 21  N2=1.0220 78  total=1.3171
mx_burn.s       H2O=0.0234 2   O2=0.2717 21  N2=1.0220 78  total=1.3171
sp_fuel.s       CH4O=0.0330
mx_fuel.s       CH3OH=0.0330 100  total=0.0330 :F
pefc.a          CO2=0.2970 48  H2=0.1300 21  H2O=0.1436 23  N2=0.0484 8   total=0.6190
cp_anode        CO=0.0002 0   CO2=0.2968 48  H2=0.1298 21  H2O=0.1437 23  N2=0.0484 8
                total=0.6190
sp_anode        CO=0.0002 0   CO2=0.2968 48  H2=0.1298 21  H2O=0.1437 23  N2=0.0484 8
                total=0.6190
mx_burn         CO2=0.2970 16  H2O=0.2970 16  O2=0.2067 11  N2=1.0705 57  total=1.8711
mx_fuel         CO2=0.3300 17  H2O=0.3630 19  O2=0.1571 8   N2=1.0704 56  NO=0.0001 0
                total=1.9206
form.h          CO2=0.3300 17  H2O=0.3630 19  O2=0.1572 8   N2=1.0705 56  total=1.9206
mx_cath         CO2=0.3300 6   H2O=0.9367 17  O2=0.5241 9   N2=3.8336 68  total=5.6244
mx_anode.s      CO2=0.3300 6   H2O=0.9367 17  O2=0.5241 9   N2=3.8336 68  total=5.6244
sp_anode.s      total=0.00e+00
mx_anode        CO2=0.3300 6   H2O=0.9367 17  O2=0.5241 9   N2=3.8336 68  total=5.6244
gt_1            CO2=0.3300 6   H2O=0.9367 17  O2=0.5241 9   N2=3.8336 68  total=5.6244
cond_1          CO2=0.3300 6   H2O=0.7477 14  O2=0.5241 10  N2=3.8336 71  total=5.4353
sp_h2o.s        H2Oc=0.2069 100  total=0.2069
h2o             H2O=0.00e+00  H2Oc=0.2069
mx_h2o.s        H2O=0.00e+00  H2Oc=0.2069
water_tank      H2O=0.00e+00  H2Oc=51.699
pump_water      H2O=0.00e+00  H2Oc=51.699
pefc.cool       H2O=0.00e+00  H2Oc=51.699
mx_h2o          H2O=0.00e+00  H2Oc=51.906
sp_wat          H2O=0.00e+00  H2Oc=51.510
hx_preh.h       H2O=0.00e+00  H2Oc=51.510
hx_rej.h        H2O=0.00e+00  H2Oc=51.510
mx_cond.s       H2O=0.00e+00  H2Oc=51.510
cond_1.s        H2O=0.00e+00  H2Oc=0.1891
mx_cond         H2O=0.00e+00  H2Oc=51.699
water_tank.cyc  H2O=0.00e+00  H2Oc=51.699
sp_wat.s        H2O=0.00e+00  H2Oc=0.3960
hx_cool         H2O=0.2563  H2Oc=0.1397
wat.cyc         H2O=0.2563  H2Oc=0.1397
air_int         H2O=0.6262 2   O2=7.2789 21  N2=27.383 78  total=35.288
fan_int         H2O=0.6262 2   O2=7.2789 21  N2=27.383 78  total=35.288
cp_air.cool     H2O=0.6262 2   O2=7.2789 21  N2=27.383 78  total=35.288
air_cond        H2O=0.6262 2   O2=7.2789 21  N2=27.383 78  total=35.288
fan_cond        H2O=0.6262 2   O2=7.2789 21  N2=27.383 78  total=35.288
cond_1.cool     H2O=0.6262 2   O2=7.2789 21  N2=27.383 78  total=35.288
air_rej         H2O=3.1309 2   O2=36.395 21  N2=136.91 78  total=176.44
fan_rej         H2O=3.1309 2   O2=36.395 21  N2=136.91 78  total=176.44
hx_rej          H2O=3.1309 2   O2=36.395 21  N2=136.91 78  total=176.44
```

```
                    output of model powers

  model           input        loss         prod          cons
                   (W)          (W)          (W)           (W)

cp_air          0.0000e+00  0.0000e+00  0.0000e+00  1.8467e+04
pump_fuel       0.0000e+00  0.0000e+00  0.0000e+00  3.8913e+00
pefc            0.0000e+00  0.0000e+00  1.0141e+05  0.0000e+00
cp_anode        0.0000e+00  0.0000e+00  0.0000e+00  9.6597e+02
gt_1            0.0000e+00  0.0000e+00  1.6062e+04  0.0000e+00
pump_water      0.0000e+00  0.0000e+00  0.0000e+00  1.2750e+02
water_tank      0.0000e+00  8.0413e+01  0.0000e+00  0.0000e+00
wat             8.1577e-04  0.0000e+00  0.0000e+00  0.0000e+00
fan_int         0.0000e+00  0.0000e+00  0.0000e+00  5.5113e+02
fan_cond        0.0000e+00  0.0000e+00  0.0000e+00  5.5113e+02
fan_rej         0.0000e+00  0.0000e+00  0.0000e+00  2.7556e+03
totals          8.1577e-04  8.0413e+01  1.1747e+05  2.3422e+04

netprod         9.4049e+04
netinput       -8.0412e+01
```

Distribution for ANL-98/8

Internal:

R. K. Ahluwalia (10)
S. K. Bhattacharyya
M. K. Butler
E. D. Doss
P. J. Finck
H. K. Geyer (10)
L. R. Johnson
M. Krumpelt
R. Kumar (10)
J. J. Laidler
R. P. Larsen
J. F. Miller
K. M. Myles
V. J. Novick
M. Q. Wang
T. B. Yafe
TIS Files

External:

DOE-OSTI (2)
ANL-E Library
ANL-W Library
J. M. Bentley, Arthur D. Little, Inc., Cambridge, MA
L. Cataquiz, USDOE, Office of Transportation Technologies, Washington, DC
S. G. Chalk, USDOE, Office of Transportation Technologies, Washington, DC
P. Davis, USDOE, Office of Transportation Technologies, Washington, DC
W. D. Ernst, Plug Power L.L.C., Latham, NY
J. Ferrell, USDOE, Office of Transportation Technologies, Washington, DC
R. J. Fiskum, USDOE, Office of Building Energy Research, Washington, DC
J. Garbak, USDOE, Office of Transportation Technologies, Washington, DC
S. Gronich, USDOE, Office of Solar Energy Conversion, Washington, DC
T. J. Gross, USDOE, Office of Transportation Technologies, Washington, DC
H. J. Hale, USDOE, Office of Transportation Technologies, Washington, DC
K. L. Heitner, USDOE, Office of Transportation Technologies, Washington, DC
D. L. Ho, USDOE, Office of Transportation Technologies, Washington, DC
G. Joy, U.S. Department of Commerce, Washington, DC
K. Kinoshita, Lawrence Berkeley Laboratory, Berkeley, CA
R. S. Kirk, USDOE, Office of Transportation Technologies, Washington, DC
R. A. Kost, USDOE, Office of Transportation Technologies, Washington, DC
J. Larkins, Georgetown Energy Programs, Washington, DC
J. Milliken, USDOE, Office of Transportation Technologies, Washington, DC
D. J. Nelson, Virginia Polytechnic Institute and State University, Blacksburg, VA
J. M. Ohi, National Renewable Energy Laboratory, Golden, CO
P. G. Patil, USDOE, Office of Transportation Technologies, Washington, DC
T. Rehg, Allied-Signal, Torrance, CA
V. P. Roan, University of Florida, Palm Beach Gardens, FL
R. Ross, Energy Partners, West Palm Beach, FL
C. Sloane, GM Research and Development Center, Warren, MI
S. Swathirajan, General Motors Research Laboratories, Warren, MI
C. E. Thomas, Directed Technologies, Inc., Arlington, VA
N. E. Vanderborgh, Los Alamos National Laboratory, Los Alamos, NM
R. White, University of South Carolina, Columbia, SC
K. Wipke, National Renewable Energy Laboratory, Golden, CO

X