SCoP Language Summary

August 7, 1993

1 General Characteristics

SCoP uses a text-based language to define a simulation model, its variables, and the numerical method(s) used to solve the model equations. The SCoP language is block oriented and there are two types of blocks: definition blocks for variables and constants and equation blocks to define the model structure. In addition, the language contains directives to specify numerical methods and other control actions.

The format of the model file is mostly up to the user since there are only a few specific rules that must be followed. All symbols are delimited by "white space" (spaces, tabs, or end of line) except in lists, where they are delimited by commas. Devices such as indents, tabbed columns, and blank lines can thus be used freely to define a style that is easy to read and understand.

All definition blocks must come first in the model source file before the equation blocks; within each group (definition and equation), the block order is arbitrary. All blocks are optional except where indicated.

In the descriptions below, pairs of angle brackets and their contents (e.g. `<name>`) are to be replaced by strings or numbers supplied by the user. Square brackets indicate optional directives (except when they indicate array subscripts).

2 Variable Declaration Blocks

Names for variables and constants can be defined by the user according to the following rules:

- Names can contain from 1 to 20 characters. Generally, longer and more descriptive names are recommended to make model files easy to read. There are, however, some circumstances where compound names are constructed from variable names and extensions (e.g. submodels and sensitivity calculations). In these cases, shorter variable names are advisable to avoid compound names longer than 20 characters.

- The following characters are legal in names: upper and lower case alphabet, numerals 0 through 9, and the underbar. The first character in a name must be alphabetic.

Cases are distinguished; `TotalNa` is different from `totalna`, for example. Block names, directives, and other key words in SCoP are defined with all capital letters to help distinguish them from user-defined names.

- Reserved words defined in the SCoP language must not be redefined by the user. The list of reserved words is given in a later section and includes block names, directives, logical operations, and names of library functions.

```
PARAMETER
{
  <variable_name> [= <value> (<units_string>)]
}
```

Defines parameters whose value can be changed within SCoP and optionally assigns values and units. A default value of zero is inserted if none is specified.

```
CONSTANT
{
  <variable_name> [= <value> (<units_string>)]
}
```

Defines constants whose value *cannot* be changed within SCoP and optionally assigns values and units. A default value of zero is inserted if none is specified.

```
STEPPED
{
  <variable_name> = <value_list> [(<units_string>)]
}
```

Defines a single constant that has a list of from two to five values in its "value string". Ordinary constants can be changed into stepped constants at run time, but only one constant can be stepped at a time.

```
INDEPENDENT
{
  <name> FROM <value> TO <value> WITH <# breakpoints> [START <value>]
 [(<units_string>)]
}
```

There must be *exactly* one INDEPENDENT block and one variable defined – no more and no less. As with state variables, a CONSTANT `<name0>` is created to hold the starting value, set to the FROM value. An additional CONSTANT `<delta_name>` is also created as an increment for numerical solvers (e.g. a time step for integrators) and given a default value equal to the spacing of the breakpoints. Another CONSTANT `<maxerr>` is created

for specifying the error limit for integrators with automatic time step adjustment. The FROM-TO values are used for the min and max of the scale when plotting the independent variable. Model solution begins with the independent variable set to the START value, if one is specified. If a START value is not specified, the independent variable begins with the FROM value. A SWEEP directive can be inserted in the INDEPENDENT block to change the effective independent variable; the SWEEP directive is described in a following section. Note that the minimum model file that produces a running SCoP program consists of one line:

```
INDEPENDENT { t FROM 0 TO 1 WITH 1 }
```

```
STATE
{
  <name> [FROM <number> TO <number> START <number> (<units_string>)]
}
```

Solvers in the SCoP library assign values to STATE variables so they should appear only in state equations (denoted with a tilde ~ at the beginning of the line) and on the right-hand side of other forms of equations. If `<name>` is followed by an integer in square brackets (e.g. `<name>[n]`), the line defines a vector variable containing n double-precision values, numbered from 0 to n-1. Definitions of state variables may also appear within other blocks, where they are treated as local variables. The optional FROM-TO range is used only by the SCoP graphics routines to determine the plot scale when the state variable is selected for plotting, i.e. the range does not affect calculations. If a FROM-TO range is not supplied, default values of 0 and 1 are assumed. If the optional START (initial) value is not supplied, a default initial value of zero is supplied. Two additional variables are created to accompany each state variable: a CONSTANT `<name0>` for the initial value of `<name>` and an ASSIGNED `<Dname>` for the derivative.

```
ASSIGNED
{
  <name> [FROM <number> TO <number> (<units_string>)]
}
```

ASSIGNED variables have values calculated and assigned to them by appearing on the left-hand side of ordinary assignment statements (i.e. not state equations). If `<name>` is followed by an integer in square brackets (e.g. `<name>[n]`), the line defines a vector variable containing n double-precision values. The optional FROM-TO range is used only by the SCoP graphics routines to determine the plot scale when the assigned variable is selected for plotting, i.e. the range does not affect calculations. If a FROM-TO range is not supplied, default values of 0 and 1 are assumed.

Constants and variables may only be declared once; multiple declarations are considered to be an error.

3 Equation Block Definitions

Unless otherwise indicated, there may be multiple blocks of the following types. In the blocks below, `<statement list>` denotes an arbitrary number of statements including

- Equations assigning values to assigned variables, of the form
  `<var_name> = <expression>`
- Directive statements (listed below)
- Conditional sub-blocks
- Lists of statments enclosed in braces. This is functionally equivalent to
  `IF (1) {...}`

`<expression>` refers to a combination of variable and constant names, literal constants, the arithmetic operators defined below, parentheses, and function calls. SCoP expressions follow the rules used by most computer languages; for specifics, refer to a description of the C language.

```
INITIAL
{
  <statement list>
}
```

The optional INITIAL block is run once at the beginning of each simulation run. All state variables are automatically assigned initial values supplied in their definitions, but these default values can be superceded or other one-time calculations carried out by statements in an INITIAL block. If multiple INITIAL blocks exist, their contents are concatenated.

```
BREAKPOINT
{
  <statement list>
}
```

This is the main block specifying model calculations. Statements within the BREAKPOINT block are executed once for each output breakpoint. Typical statements appearing in the BREAKPOINT block are SOLVE directives, conditional sub-blocks, and assignment statements for assigned variables.

```
TERMINAL
{
  <statement list>
}
```

Statements in the optional TERMINAL block may be executed once at the end of a simulation run through a selection in the End-Run menu of SCoP.

```
LINEAR <block_name>
{
  [ <statement list> ]
  ~ <expression> = <expression>
  [ <statement list> ]
}
```

A LINEAR block contains linear algebraic equations for state variables, i.e. each `<expression>` is made up of a sum of terms containing no more than one state variable as a multiplier. Equations marked by the tilde (~) in the LINEAR block are parsed so that state variables to be solved for can appear on either or both sides of such equations. Additional terms can be added to either side of a state variable equation in a LINEAR block by following the equation line with a statement of the form

```
~+ <expression> = <expression>
```

The expressions on either side of the equal sign are added to their respective sides of the preceeding state equation. Assignments and equations can be mixed in any order and also occur in conditional sub-blocks. Each LINEAR block must be accompanied by a SOLVE directive in some other block, usually an EQUATION block. Solvers require the number of independent equations in a LINEAR block to be equal to the number of state variables solved for.

```
NONLINEAR <block_name>
{
  [ <statement list> ]
  ~ <expression> = <expression>
  [ <statement list> ]
}
```

A NONLINEAR block contains nonlinear algebraic equations for state variables. Equations marked by the tilde in the NONLINEAR block are parsed so that state variables to be solved for can appear on either or both sides of such equations. Additional terms can be added to either side of a state variable equation in a NONLINEAR block by following the equation line with a statement of the form

```
~+ <expression> = <expression>
```

The expressions on either side of the equal sign are added to their respective sides of the preceeding state equation. Each NONLINEAR block must be accompanied by a SOLVE directive in the EQUATION block. Solvers require the number of independent equations in a NONLINEAR block to be equal to the number of state variables to be solved for.

```
DERIVATIVE <block_name>
{
```

```
  [ <statement list> ]
  var' = <expression>
  [ <statement list> ]
}
```

A DERIVATIVE block defines a group of ordinary differential equations to be solved together by the same numerical method and with the same time step. The numerical solvers in the SCoP library are designed for sets of first order differential equations. Higher order equations may be included, however; SCoP reduces these to sets of first order equations by creating the state variables and equations for lower derivatives. Higher order derivatives are indicated by repeating the *single* prime symbol: y'' is the second derivative, y''' the third derivative, etc. (i.e. do *not* use the double prime symbol). Each differential equation must be rearranged to isolate the highest derivative on the left-hand side of the equals sign with all other terms and factors moved to the right-hand side.

As indicated in the standard form above, ordinary assignment statements may be included in DERIVATIVE blocks, before or after or mixed in with differential equations. Nonlinear algebraic equations may also be included (marked with a tilde as in the NONLINEAR block) and will be solved along with the differential equations.

Initial conditions for the state variables in DERIVATIVE blocks can be supplied by any one of four methods:

1. Starting values can be included with the declaration of the state variables by adding `START <value>` after the FROM-TO range in the STATE block. For a variable named y SCoP creates a constant named y0 for this starting value.

2. The constant y0 can be declared in the CONSTANT block, along with its value, by the user.

3. If neither of the preceeding two methods are utilized, SCoP will still create the constant y0 automatically and give it a value of zero. This value can be changed interactively at run time.

4. The INITIAL block can also be used to assign a value to y0. This method is most often used when calculations are necessary to arrive at the starting value.

Each DERIVATIVE block must be accompanied by a SOLVE directive in the EQUATION block.

```
PARTIAL <block_name>
{
  [ <statement list> ]
  ~ var' = f_val * DEL2(var) + g_val
  [ <boundary conditions> ]
  [ <statement list> ]
}
```

The PARTIAL block defines a parabolic partial differential equation, i.e. a partial differential equation with a first-order time derivative term and a second-order space derivative term (no mixed derivative term or first-order space derivative term). var, f_val, and g_val are vector variables of equal length. f_val and g_val contain the coefficients in the partial differential equation.

A boundary condition on the state variable or its derivative must be supplied at each spatial limit. The default boundary conditions are zero spatial derivatives at both limits. Explicit boundary conditions are indicated by one of the following forms,

```
~ var[<limit>] = <expression>
```

for values of the state variable itself (Dirichlet or first kind) or

```
~ DEL var[<limit>] = <expression>
```

for values of the derivative (Neumann or second kind). In either of these forms, `<limit>` takes on one of the values `FIRST` (equivalent to x = 0) or `LAST`.

Each PARTIAL block must be accompanied by a matching SOLVE directive in the EQUATION block.

```
DISCRETE <block_name>
{
  [ <statement list> ]
  var1 = f(var1@1,var1@2,...,var2@1,...)
  [ <statement list> ]
}
```

A DISCRETE block defines a group of discrete equations, where the current value of var1 is a function of variables at previous time steps. `<var@n>` is the stored value of `<var>` at the end of the nth previous step (`var@0` is not allowed). Histories of each variable in a DISCRETE block are automatically maintained for the required number of time steps. Variables used in DISCRETE equations must be declared as state variables and may be vector variables as well. Each DISCRETE block must be accompanied by a matching SOLVE directive in the EQUATION block.

```
KINETIC <block_name>
{
  [<statement list>]
  ~ <stoic> <var> [ + ...] <-> ... (kf, kr)
  [<statement list>]
}
```

A KINETIC block contains a set of equations describing reactions of the chemical reaction type. On each side of the double arrow `<->` in a reaction is a sum of terms consisting

of an optional stoichiometric number (small integer) multiplying any type of SCoP variable (CONSTANT, ASSIGNED, STATE, INDEPENDENT, STEPPED, or SWEEP). The space between the number and the variable is optional. kf and kr following each reaction are forward and reverse rate coefficients for the reaction and can be arbitrary expressions. The effect of any type of variable other than a state variable is to multiply one of the rate coefficients by the variable raised to the power of its stoichiometric number. If the variable appears on the left-hand side of the reaction, it multiplies the forward rate coefficient and if it is on the right-hand side it multiplies the reverse rate coefficient.

Three additional types of equations are also possible in KINETIC blocks.

`~ <var> -> (<rate expression>)`

represents a flux out of a state with the specified rate coefficient.

`~ <var> << (<flux expression>)`

represents a flux into a state.

`CONSERVE <state expression> = <constant expression>`

represents a form of algebraic equation (such as a conservation law) that is solved with the equations from the specified reactions. Each CONSERVE equation added to a KINETIC block results in the deletion of an appropriate equation from the reaction set (to keep the total number of equations constant for the numerical solvers). `<state expression>` must be a linear equation in the KINETIC state variables. Reactions in a KINETIC block are transformed into ordinary differential equations for the state variables that can be integrated via a SOLVE directive in the EQUATION block. Solvers require that the number of state equations in a KINETIC block be equal to the number of state variables in the block.

Immediately following each reaction equation, the variables `f_flux` and `b_flux` contain the unidirectional fluxes of the reaction from left to right ("forward") and right to left ("backward"), respectively. If these flux values are needed, they must be stored by the user before the next reaction equation since the same variables are used for all reaction equations. Since the unidirectional fluxes are computed from the values of the state variables at the *beginning* of each integration step, the fluxes should be used with caution when there are large changes in each time step.

```
PROCEDURE <proc_name> [(<argument list>)]
{
  <statement list>
}
```

A PROCEDURE block defines a subroutine of user-written code to be called elsewhere in the file. A PROCEDURE block is called in the manner of a subroutine call in C, i.e. by giving the PROCEDURE name with a matching list of arguments. Note that arguments are passed by value only, so that numerical results can be returned only in global variables. PROCEDURE blocks do not return values.

```
FUNCTION <func_name> [(<argument list>)]
{
```

```
  <statement list>
  <func_name> = <expression>
}
```

A FUNCTION block contains user-written code producing a double-precision numeric result. A FUNCTION block is used in the manner of a function in C, i.e. by naming the FUNCTION with its matching list of arguments in an expression. Note that arguments are passed by value only. As in C, the FUNCTION block must contain at least one statement assigning a value to `<func_name>`.

```
COMMENT
  <text>
ENDCOMMENT
```

Text in a COMMENT block is ignored and does not contribute to model equations. Any text on a line following a colon is also treated as a comment and ignored.

```
VERBATIM
  <C code>
ENDVERBATIM
```

The VERBATIM block permits C code to be inserted anywhere in model calculations without modification by the SCoP preprocessor.

4 Conditional Sub-blocks

As in C, conditional sub-blocks may be nested. Note that unlike the C language, the curly braces are not optional in conditional sub-blocks in SCoP and must be used even if they surround only a single statement.

```
IF [<unary_logical_operator>] <expression>
        [<binary_logical_operator> <expression>]
{
<statement list>
}
[ELSE
{
<statement list>
}]

WHILE [<unary_logical_operator>] <expression>
        [<binary_logical_operator> <expression>]
{
<statement list>
}
```

```
FROM <index> = <expression> TO <expression> [BY <positive increment>]
{
<statement list>
}
```

5 Directive Definitions

`TITLE <title string>`
Text from the first non-blank character to the end of the line is used as the name of the model in SCoP displays. Subsequent TITLE lines are ignored. If there is no TITLE line, the name of the model is an empty string. Two characters that might be used in TITLE lines receive special treatments in C and have to be preceeded by a backslash: the double quote must be entered as `\"` and the backslash must be written as `\\`.

`DEFINE <string> <integer>`
This directive defines a macro for replacement prior to processing.

`SWEEP <var> FROM <value> TO <value> WITH <# breakpoints> [(<units_string>)]`

This directive is used inside the INDEPENDENT block to change the *effective* independent variable. When this directive is used, the solver generates a solution to the final value of the independent variable for each value of the SWEEP variable.

`LOCAL <variable list>`
This directive defines double-precision variables for use only within the block containing the directive. LOCAL variables do not appear in the variable file. The LOCAL directive must always be the first thing in a block. LOCAL directives not enclosed in a block declare variables that can be used in any block, but do not appear in the variable file.

`SOLVE <block_name> [METHOD <method_name>]`
SOLVE directives accompany equation blocks of types requiring numerical solution: LINEAR, NONLINEAR, DERIVATIVE, KINETIC, PARTIAL. Each type has a default solution method that is used if the METHOD is not specified. SOLVE directives normally appear in the EQUATION block.

`RESET`
This directive re-initializes multi-step integrators to handle discontinuities in the derivative functions. At a discontinuity (e.g. the application of a step as a driving function) the previous history of the solution is no longer a good indicator of the future behavior of the solution and the numerical method must be restarted. The RESET directive should be used in some kind of logical structure to determine when it is to be executed.

`PLOT <vert_var_list> VS <horiz_var_name>`
The PLOT directive sets up the initial choices of variables to be plotted on the horizontal and vertical axes in the graphic mode or displayed in columns in the tabular mode. `<vert_var_list>` can be either a single variable or a list of up to five variables, separated by commas. The selections in the PLOT directive can be overridden by the user at run time from the Output Format Menu. If there is no PLOT directive, SCoP uses as default

selections the independent variable for the horizontal axis and the first state variable for the vertical axis. Single elements of vector variables may be used as variable names, e.g. `V[5]`. If the name of a vector variable is used without a specific index value (e.g. V where V is a vertor variable), the default selection is `V[size]`. This form is used to call for the special plot of one vector variable against another of the same length.

`SENS <parameter list>`

Generates one call to the sensitivity equations evaluation function for each parameter in the list and can be used within the LINEAR, NONLINEAR, DERIVATIVE, and KINETIC blocks [1]. SENS also creates the following additional variables for each parameter in the list:

`S_<state>_<param>` State variable for the parameter variations

`DS_<state>_<param>` Derivative for the state variable if a trajectory sensitivity is being calculated.

`U_<parameter>` The relative uncertainty of the parameter value (default = 0.05)

`EP_<state>_<parameter>` Envelope for positive parameter uncertainty

`EM_<state>_<parameter>` Envelope for negative parameter uncertainty

The envelope variables contain the values of the state variable with the parameter shifted upward and downward by the relative uncertainty.

6 Operators

~ Introduces linear equations, nonlinear equations, kinetic reactions, and partial differential equations.

`~+` Add the following equation of the form `<expression> = <expression>` to the last LINEAR or NONLINEAR state equation defined. Note that if one expression is 0, only one side of the equation is changed.

@n Postfix operator (n is an integer) creates a local variable for the value of the previous variable n steps earlier

: Introduces in-line comment

6.1 Arithmetic Operators (infix)

+ Addition

- Subtraction

* Multiplication

[1] Note that SENS can, at present, only be used with the *non-default* integrators in the KINETIC block

/ Division

^ Exponentiation

6.2 Logical Operators (infix except for NOT)

Note that these logical operators are in the style of the C language for consistency with VERBATIM blocks.

&& Logical "and"

|| Logical "inclusive or"

! Logical "not" (prefix)

== Logical equality test

!= Logical "not equals"

> Greater than

< Less than

>= Greater than or equals

<= Less than or equals

7 Reserved Words

acos	fmod	revhyperbol
asin	force	revsawtooth
ASSIGNED	FROM	revsigmoid
atan	FUNCTION	sawtooth
atan2	gauss	SENS
b_flux	harmonic	set_seed
BY	hyperbol	sigmoid
ceil	IF	sin
COMMENT	INDEPENDENT	sinh
CONSERVE	INITIAL	SOLVE
CONSTANT	KINETIC	spline
cos	lag	squarewave
cosh	LINEAR	sqrt
DEFINE	LOCAL	START
deflate	log	STATE
DEL	log10	step
DERIVATIVE	maxerror	STEPPED
derivs	METHOD	SWEEP
DISCRETE	NONLINEAR	tan
ELSE	normrand	tanh
ENDCOMMENT	PARTIAL	TERMINAL
ENDVERBATIM	perpulse	threshold
EQUATION	perstep	TITLE
erf	PLOT	TO
exp	poisrand	VERBATIM
expfit	poisson	VS
exprand	pow	WHILE
f_flux	prterr	WITH
fabs	pulse	
factorial	ramp	
floor	random	
	RESET	

8 C Math Library Functions

The functions described here are intrinsic (built-in) functions which are provided by the C language math library. These functions may be utilized in the equations the modeler uses to describe his SCoP model. For each function, we provide a synopsis describing the function, the arguments it expects, and the value which it returns. With the exception of **abs()**, all of these functions return double-precision floating point values and expect arguments which are double-precision floating point values.

acos(x) - compute the arc cosine : Returns the arc cosine of its argument ($\arccos x$). The argument must be in the range -1.0 to 1.0.

asin(x) - compute the arc sine : Returns the arc sine of its argument ($\arcsin x$). The argument must be in the range -1.0 to 1.0.

atan(x) - compute the arc tangent : Returns the arc tangent of its argument ($\arctan x$).

atan2(y,x) - compute the arc tangent of y/x : Returns the arc tangent of the ratio y/x ($\arctan y/x$).

ceil(x) - round upwards : Returns the smallest integral value greater than or equal to its argument ($\lceil x \rceil$).

cos(x) - compute the cosine : Returns the cosine of its argument ($\cos x$). The angle is expressed in radians.

cosh(x) - compute the hyperbolic cosine : Returns the hyperbolic cosine for a real argument ($\cosh x$).

exp(x) - compute the exponential function : Returns the value of e raised to the argument power (e^x).

fabs(x) - compute the absolute value of a floating point number : Returns the absolute value of its argument ($|x|$).

floor(x) - round downwards : Returns the largest integral value less than or equal to its argument ($\lfloor x \rfloor$).

fmod(x,y) - compute x modulo y : Returns the value of x modulo y ($x \bmod y$), the remainder resulting from x/y.

log(x) - compute the natural logarithm : Returns the natural logarithm of its argument ($\ln x$).

log10(x) - compute the base 10 logarithm : Returns the base 10 logarithm of its argument ($\log_{10} x$).

pow(x,y) - compute x raised to the y power : Returns the value of x raised to the power of y (x^y).

sin(x) - compute the sine : Returns the sine of its argument ($\sin x$). The angle is expressed in radians.

sinh(x) - compute the hyperbolic sine : Returns the hyperbolic sine for a real argument ($\sinh x$).

sqrt(x) - compute the square root : Returns the positive square root of its argument ($\sqrt{x}$).

tan(x) - compute the tangent : Returns the tangent of its argument ($\tan x$). The angle is expressed in radians.

tanh(x) - compute the hyperbolic tangent : Returns the hyperbolic tangent for a real argument ($\tanh x$).

9 SCoP Library Functions

Many of the functions which make up the SCoP library are used by SCoP internally and are of no direct interest to the user. However, there are some functions which the SCoP library provides which are of a more general nature and which may be utilized by the user's model.

9.1 Program Interaction Functions

prterr(message_string) - print message and wait : Takes a single argument which is a message string (e.g. *message_string* might be `"This is another fine mess\n"`). **prterr()** then prints that message on the screen, followed by this prompt to the user:

```
Press <Enter> to continue
```

and waits until the user presses the `<Enter>` key (also sometimes referred to as `Return`, `<CR>`, or $\hookleftarrow$, it is the carriage return key for the terminal, PC or workstation in use).

Return Value:

None

9.2 Curve-fitting and Interpolation Functions

derivs(nbase, x, y, mesh, der) - compute second derivatives : This function computes the second derivatives at the base points for cubic spline interpolation. This function is called once prior to calling **spline()** for each interpolation (see **spline()** below). Inputs:

nbase	number of base points (an integer)
x	vector of X values at the base points (vector of double-precision floating point values)
y	vector of Y values at the base points (vector of double-precision floating point values)

Outputs:

mesh vector containing base point spacing (vector of double-precision floating point values)

der vector containing the second derivatives of the base points (vector of double-precision floating point values)

Return Value:

derivs() returns an integer which is 0 on success, and 1 if tridiagonal matrix is singular or ill-conditioned.

spline(nbase, x, y, mesh, der, x_inter) - compute cubic spline : This function evaluates the cubic spline at the point for which an interpolation is desired. Call this function once for each interpolation to be performed (Note: the function **derivs()** (see above) must be called prior to calling **spline()** for each interpolation).

Inputs:

nbase number of base points (an integer)

x vector of X values at the base points (vector of double-precision floating point values)

y vector of Y values at the base points (vector of double-precision floating point values)

mesh vector containing base point spacing (vector of double-precision floating point values)

der vector containing the second derivatives of the base points (vector of double-precision floating point values)

x_inter value at which interpolation is desired (a double-precision floating point value)

Return Value:

spline() returns the value of the cubic spline at the requested point (a double-precision floating point value).

force(t, filename) - generalized forcing function : This function reads a file containing the coordinates of a forcing function in reference data file format. The argument *filename* is a string containing the name of the file to be read (e.g. *filename* might be `"FORFUNC.DAT"`). (Remember that the backslash \ is an escape character in C strings and thus must be preceeded by a second backslash if used in directory names in the DOS operating system.) The first argument t is the value of the test variable at which the (interpolated) value of the forcing function is requested (Note: **force()** calls **derivs()** and **spline()** as necessary to accomplish the interpolation).

Return Value:

force() returns the requested value of the (interpolated) forcing function (a double-precision floating point value).

deflate(degree, coeff, root) - compute roots of a polynomial : This function finds the real roots of a polynomial with real coefficients by Newtonian deflation.

Inputs:

degree degree of the polynomial (an integer value)

coeff vector of real coefficients of the polynomial (vector of double-precision floating point values)

Outputs:

root vector of real roots of the polynomial (vector of double-precision floating point values)

Return Value:

deflate() returns the number of real roots found (an integer value).

expfit(terms, reffile, amplitude, lambda, error) - exponential fit : This function performs a fit to reference data using Handscomb's modification to Prony's method for exponential curve peeling. Only the real poles are found. The data must be sampled at even intervals and stored in reference data file format. Up to 201 data points are allowed.

Inputs:

terms maximum number of terms to try to fit (an integer value). If $terms < 0$, discard exponentials with amplitudes $< error$.

reffile a string containing the name of the file to be read (e.g. *reffile* might be `"C:\MODEL\REFDATA.DAT"`)

Outputs:

amplitude vector of pre-exponential coefficients (vector of double-precision floating point values)

lambda vector of time constants (vector of double-precision floating point values)

error standard error of estimate (a double-precision floating point value). Set to -1 if $terms > 0$

Return Value:

expfit() returns the number of exponentials in the fit to the data (an integer value).

If the function is invoked as:

```
nexp = expfit(terms, reffile, amplitude, lambda, error);
```

then the resulting function is:

$$y = amplitude[0] * \exp(lambda[0]) * t) + \ldots + amplitude[terms - 1] * \exp(lambda[terms - 1] * t)$$

if $terms > 0$, and

$$y = amplitude[0] * \exp(lambda[0]) * t) + \ldots + amplitude[nexp - 1] * \exp(lambda[nexp - 1] * t)$$

if $terms < 0$ ($nexp \leq terms$).

9.3 General Modeling Functions

hyperbol(x, max, K) - compute an hyperbolic function : This function computes the value of an hyperbolic function, given the values of two constant parameters.

Inputs:

x	the independent variable (a double-precision floating point value)
max	the maximum value of the hyperbolic function (a double-precision floating point value)
K	the value of x for which the function value is $max/2$ (a double-precision floating point value)

Return Value:

hyperbol() returns the value of the hyperbolic function (a double-precision floating point value).

revhyperbol(x, max, K) - compute a reverse hyperbolic function : This function computes the value of a reverse hyperbolic function, given the values of two constant parameters.

Inputs:

x	the independent variable (a double-precision floating point value)
max	the maximum value of the hyperbolic function (a double-precision floating point value)
K	the value of x for which the function value is $max/2$ (a double-precision floating point value)

Return Value:

revhyperbol() returns the value of the reverse hyperbolic function (a double-precision floating point value).

sigmoid(x, max, K, n) - compute a sigmoidal function : This function computes the value of a sigmoidal function of one variable, given the values of three constants in the function.

Inputs:

x	the independent variable (a double-precision floating point value)
max	the maximum value of the sigmoidal function (a double-precision floating point value)
K	the value of x for which the function value is $max/2$ (a double-precision floating point value)
n	parameter controlling the steepness of the sigmoid curve (a double-precision floating point value); it is the power to which x and K are raised

Return Value:

sigmoid() returns the value of the sigmoidal function (a double-precision floating point value).

revsigmoid(x, max, K, n) - compute a reverse sigmoidal function : This function computes the value of a reverse sigmoidal function of one variable, given the values of three constants in the function.

Inputs:

x	the independent variable (a double-precision floating point value)
max	the maximum value of the sigmoidal function (a double-precision floating point value)
K	the value of x for which the function value is $max/2$ (a double-precision floating point value)
n	parameter controlling the steepness of the sigmoid curve (a double-precision floating point value); it is the power to which x and K are raised

Return Value:

revsigmoid() returns the value of the reverse sigmoidal function (a double-precision floating point value).

lag(var, curt, lagt, vsize) - generalized lag function : This function provides a generalized lag facility for the SCoP modeler, but use of it requires more than a passing knowledge of C language syntax (this 'limitation' will be addressed in a future version of SCoP). The **lag()** function has two independent variables, *var* and *curt*, where *curt* is the current value of the monotonically increasing independent variable of the

model (usually time) and *var* is a pointer to an independent variable whose value at $curt - lagt$ is desired. **lag()** is currently able to handle up to 50 lagged model variables.

Inputs:

var	pointer to the independent variable whose value at $curt - lagt$ is desired (a pointer to a double-precision floating point value)
curt	current value of the monotonically increasing independent variable of the model [usually time] (a double-precision floating point value)
lagt	difference between *curt* and the previous value of *curt* for which we desire the value of *var* (a double-precision floating point value)
vsize	the dimension of *var* if *var* is a vector; otherwise ≤ 1 (an integer value)

Return Value:

lag() returns a pointer to the double-precision floating point value of the variable *var* at $curt - lagt$ if $curt \geq lagt$, or the `NULL` pointer if $curt < lagt$.

9.4 Forcing Functions

threshold(x, limit, mode) - threshold function : This function provides a switch between active and inactive state, based on a threshold value.

Inputs:

x	the variable to be tested (a double-precision floating point value)
limit	the threshold switch point (a double-precision floating point value)
mode	a string containing either `"min"` or `"max"`, indicating whether *limit* is to be treated as a minimum or a maximum value for switching to the active state

Return Value:

The return from **threshold()** is an integer value. If *mode* is `"min"`, **threshold()** returns 0 (inactive state) if $x < limit$, and returns 1 (active state) if $x \geq limit$. If *mode* is `"max"`, **threshold()** returns 0 (inactive state) if $x > limit$, and returns 1 (active state) if $x \leq limit$. **threshold()** returns -1 if there is an error in its arguments.

harmonic(t, period, amplitude, phase) - harmonic function : This function generates a sine wave of specified period, amplitude, and phase.

Inputs:

t	the independent variable [usually time] (a double-precision floating point value)

period the length of each cycle (a double-precision floating point value)

amplitude the positive amplitude of the waveform (a double-precision floating point value)

phase the phase shift of the waveform, in periods (a period is 360 degrees or 2π radians; a double-precision floating point value)

Return Value:

harmonic() returns the value of the harmonic function (a double-precision floating point value).

squarewave(t, period, amplitude) - square wave function : This function generates a square wave of specified period and amplitude.

Inputs:

t the independent variable [usually time] (a double-precision floating point value)

period the length of each cycle, including positive and negative phases (a double-precision floating point value)

amplitude the positive amplitude of the waveform (a double-precision floating point value)

Return Value:

squarewave() returns the value of the squarewave function (a double-precision floating point value), which is positive *amplitude* for the first half of the cycle and negative *amplitude* for the second half of the cycle.

sawtooth(t, period, amplitude) - sawtooth function : This function generates a sawtooth wave (periodic ramp) of specified period and amplitude.

Inputs:

t the independent variable [usually time] (a double-precision floating point value)

period the duration of the ramp incline for each cycle (a double-precision floating point value)

amplitude the ramp height for each cycle (a double-precision floating point value)

Return Value:

sawtooth() returns the value of the sawtooth function (a double-precision floating point value).

revsawtooth(t, period, amplitude) - revsawtooth function : This function generates a reverse sawtooth wave (periodic declining ramp) of specified period and amplitude.

Inputs:

t	the independent variable [usually time] (a double-precision floating point value)
period	the duration of the ramp decline for each cycle (a double-precision floating point value)
amplitude	the ramp height for each cycle (a double-precision floating point value)

Return Value:

revsawtooth() returns the value of the reverse sawtooth function (a double-precision floating point value).

ramp(t, lag, height, duration) - ramp function : This function generates a ramp function of specified height and duration.

Inputs:

t	the independent variable [usually time] (a double-precision floating point value)
lag	value of *t* at which ramp begins (a double-precision floating point value)
height	the ramp height (a double-precision floating point value)
duration	the duration of the ramp incline (a double-precision floating point value)

Return Value:

ramp() returns the value of the ramp function (a double-precision floating point value).

pulse(t, lag, height, duration) - pulse function : This function generates a single pulse of specified height and duration.

Inputs:

t	the independent variable [usually time] (a double-precision floating point value)
lag	the value of *t* at which pulse begins (a double-precision floating point value)
height	the pulse height (a double-precision floating point value)
duration	the duration of the pulse (a double-precision floating point value)

Return Value:

pulse() returns *height* during the pulse interval and zero otherwise (a double-precision floating point value).

perpulse(t, lag, height, duration, delay) - periodic pulse function : This function generates periodic pulses of specified height, duration, and delay between pulses.

Inputs:

t	the independent variable [usually time] (a double-precision floating point value)
lag	the value of t at which first pulse begins (a double-precision floating point value)
height	the pulse height (a double-precision floating point value)
duration	the duration of the pulse (a double-precision floating point value)
delay	the delay between pulses (a double-precision floating point value)

Return Value:

perpulse() returns *height* during the pulse interval and zero otherwise (a double-precision floating point value).

step(t, jumpt, jump) - step function : This function generates a single jump of specified height in a constant function.

Inputs:

t	the independent variable [usually time] (a double-precision floating point value)
$jumpt$	the value of t at which the step occurs (a double-precision floating point value)
$jump$	the step height (a double-precision floating point value)

Return Value:

step() returns zero before the step is taken ($t < jumpt$) and it returns $jump$ after the step is taken ($t \geq jumpt$). The return from **step()** is a double-precision floating point value.

perstep(t, lag, period, jump) - periodic step function : This function generates periodic jumps of specified height and duration in a constant function.

Inputs:

t	the independent variable [usually time] (a double-precision floating point value)

lag the value of t at which stepping starts (a double-precision floating point value)

period the duration of each step (a double-precision floating point value)

jump the step height (a double-precision floating point value)

Return Value:

perstep() returns zero before the first step is taken ($t < lag$) and thereafter it returns the cumulative sum of the heights of the steps already taken. The return from **perstep()** is a double-precision floating point value.

9.5 Probability Functions

erf(z) - normalized error function : This function computes the normalized error function.

$$erf(z) = \frac{2}{\sqrt{\pi}} \int_0^z \exp(-t^2)\, dt$$

Inputs:

z the value of an independent variable (a double-precision floating point value)

Return Value:

erf() returns the value of the normalized error function. This will lie in the interval between -1.0 and 1.0 and is a double-precision floating point value.

exprand() - random number from exponential distribution : This function computes a random sample from the exponential distribution.

Return Value:

exprand() returns a random number from the exponential distribution (a double-precision floating point value).

factorial(n) - factorial function : This function computes the factorial of an integer argument.

Inputs:

n the number whose factorial is desired (an integer value)

Return Value:

factorial() returns the value of the factorial ($n!$) (a double-precision floating point value).

gauss(x, mean, std_dev) - Gaussian probability density function : This function computes the value of a Gaussian probability density function at a particular x value, given the *mean* and standard deviation (*std_dev*).

Inputs:

x	the value of an independent variable (a double-precision floating point value)
mean	the mean of the Gaussian distribution (a double-precision floating point value)
std_dev	the standard deviation of the distribution (a double-precision floating point value)

Return Value:

gauss() returns the value of the probability (a double-precision floating point value).

normrand(mean, std_dev) - random number from normal distribution : This function selects a random number from the normal distribution with the specified *mean* and standard deviation (*std_dev*).

Inputs:

mean	the mean of the distribution (a double-precision floating point value)
std_dev	the standard deviation of the distribution (a double-precision floating point value)

Return Value:

normrand() returns a random number from the normal distribution (a double-precision floating point value).

poisrand(mean) - random number from Poisson distribution : This function computes a random number from the Poisson distribution with the specified *mean*.

Inputs:

mean	the mean of the distribution (a double-precision floating point value)

Return Value:

poisrand() returns an integer from the Poisson distribution (an integer value).

poisson(x, mean) - Poisson probability density function : This function computes the value of a Poisson probability density function at a particular x value, given the *mean*. Inputs:

x the value of an independent variable (an integer value)

mean the mean of the Poisson distribution (a double-precision floating point value)

Return Value:

poisson() returns the value of the probability (a double-precision floating point value).

random() - random number generator : This function selects a random number from the uniform distribution on the interval between 0 and 1. A seed can be specified for the function (see **set_seed()** below); otherwise a seed of 1 will be used.

Return Value:

random() returns the random number (a double-precision floating point value).

set_seed(n) - seed the random number function (random()) : This function sets the seed of the SCoP Library random number generator (see **random()** above).

Inputs:

n the value of the seed (an integer value)

Return Value:

None