

SGframework User's Guide

K M Kramer
Honeywell Technology Center
3660 Technology Drive
Minneapolis, MN 55418

1997

Contents

1	The SGFramework User's Manual	1
1.1	The Syntax and Grammar of the Equation Specification File	2
1.1.1	Comments	3
1.1.2	Numbers	3
1.1.3	Strings	4
1.1.4	Identifiers	5
1.1.5	Operators	7
1.1.6	Constants	8
1.1.7	Variables and Arrays	10
1.1.8	Functions	13
1.1.9	Constraint Equations	18
1.1.10	Procedures	21
1.1.11	Numerical Algorithm Parameters	30
1.2	The Syntax and Grammar of the Mesh Specification File	33
1.2.1	An Overview of the Mesh Specification File	33
1.2.2	Comments, Numbers, Identifiers and Constants	35
1.2.3	Coordinates	36
1.2.4	Points	36
1.2.5	Edges	37
1.2.6	Regions	37
1.2.7	Labels	38
1.2.8	Refinement Statements	38
1.2.9	Mesh Parameters	39
1.2.10	Element Refinement Criteria	40
1.3	Interfacing the Equation and the Mesh Specification Files	40
1.3.1	Importing an Irregular Mesh	41

1.3.2	Using Labels in Equation Specification Files	42
1.3.3	Mesh Connectivity Functions	43
1.3.4	Mesh Geometry Functions	44
1.3.5	Mesh Summation Functions	45
1.3.6	Precomputed Functions	47
1.4	SGFramework Executables	50
1.4.1	Build Script	50
1.4.2	Mesh Parser	52
1.4.3	Mesh Generator	54
1.4.4	Mesh Refiner	55
1.4.5	SGFramework Translator	56
1.4.6	Ordering Module	60
1.4.7	SGFramework Simulations	60
1.4.8	Extract Program	61
1.4.9	Group Program	62
1.4.10	Graphical Output	63

Chapter 1

The SGFramework User's Manual

The user's manual given here provides the syntax of the mesh and equation specification files, and explains the commands needed to use them. (This manual is not a tutorial; it is intended as a reference manual. The text contains an extensive tutorial on the setting up and running of simulations.)

In summary, the procedure to generate and run a simulation is given below (followed by a similar one to construct a mesh).

1. Translate the equation specification file with the SGFramework translator:

```
sgxlat filename.sg
```

2. Compile and link with the Numerical Algorithm Module:

```
sgbuild sim filename
```

3. (If necessary) order the unknown elements to reduce sparse-matrix fill:

```
order filename.top filename.prm
```

4. run the simulation:

```
filename
```

If the simulation uses an irregular mesh, you must construct it before executing the above procedure.

1. parse the mesh specification file:

```
mesh filename.sk
```

2. construct the initial mesh:

```
sggrid filename.xsk
```

3. build the mesh refinement program:

```
sgbuild ref file_ref
```

4. run the mesh refinement program:

```
file_ref
```

(Each time SGFramework opens a window to plot the mesh, it is necessary to close the window before the next line will be executed.) These commands all make use of SGFramework Executables. The SGFramework Executables are invoked from a command line. This means that Windows95 and NT users must first open an MS-DOS prompt before giving these commands to run the SGFramework Executables.

Table: Syntax of Syntax Statements

A character enclosed in single quotes represents the literal character. Hence, `'` means one literally types a comma.

The asterisk character (*) means zero or more of the item or set that preceeds it.

The plus character (+) means one or more of the item or set that preceeds it.

Items enclosed by vertical lines are optional.

Items enclosed by brackets form a set. For instance, `[A-Z]` mean one can type either an A, B, C, ..., or Z.

Items enclosed by braces form a group. Groups are treated as a single item.

Keywords appear in boldface type.

Italized text represents a previous defined entity.

In UNIX, a shell must be opened. The commands are explained in Section 1.4, on SGFramework Executables. The first parts of this manual, on the other hand, are devoted to explaining how to set up the files which are used by the SGFramework Executables.

The text to this point is full of examples of input files. The first of these is the 'Game of Life' in Section ???. This gives (and explains) the commands needed to run an existing input file (if that file does not need a mesh to be generated by SGFramework). Section ??? is the appropriate point to begin learning to use SGFramework. The examples progress through simple electrostatics problems, beginning with Section ???. The use of irregular meshes is introduced in Section ???. Section ??? is the appropriate place to begin learning about the mesh refinement program and the commands required to use it. The examples in the text are all explained where they are presented, and the easiest way to start is to read the sections which were indicated above, from the first few chapters.

Throughout this chapter there are definitions of the syntax which is appropriate for the various SGFramework commands and functions. The meaning of the syntax statements is summarized in a table of 'syntax syntax'.

This chapter is divided into four sections: The Syntax and Grammar of the Equation Specification File 1.1, The Syntax and Grammar of the Mesh Specification File 1.2, Interfacing the Equation and Mesh Specification Files 1.3, and SGFramework Executables 1.4.

1.1 The Syntax and Grammar of the Equation Specification File

This section describes the syntax and grammar of equation specification files. It is divided into twelve subsections, each of which describes a particular part of the SGFramework language. The topics covered in this section are Comments 1.1.1, Numbers 1.1.2, Strings 1.1.3, Identifiers 1.1.4, Operators 1.1.5, Constants 1.1.6, Variables and Arrays 1.1.7, Functions 1.1.8, Constraint Equations 1.1.9, Procedures 1.1.10, Conditional Statements 1.1.10, and Numerical Algorithm Parameters 1.1.11. Some of the subsections are further divided. The Functions subsection

describes internal, user-defined, and external functions, whereas the Procedures subsection describes assignment statements, single-word statements, file input/output statements, subroutine execution statements, conditional statements and looping statements.

1.1.1 Comments

Comments may be placed anywhere in the source file and are ignored by the equation specification file parser. Comments start with the `//` characters and terminate at the end of the line. It is a recommended practice to use comments to document a specification file. Proper use of comments will enable other users to easily determine the purpose of the simulation. Examples of comments can be found in the examples given in the text.

1.1.2 Numbers

Both integer and floating-point numbers are supported in equation specification files. Integers are numbers without a fractional part. Floating-point numbers contain both an integral and a fractional part.

Integers

Integers consist of an optional plus or minus sign followed by one or more digits. The range of integer numbers is dependent upon the implementation of the host system's C++ compiler. At a minimum, the integer range is guaranteed to include the numbers -32768 to 32767 inclusive. The syntax of integers is as follows.

$[[+, -]]digit$ + where *digit* is the set $[0-9]$.

The numbers 2, -5 , 12536, -32768 , and 23 are examples of allowed integers. The number 40000 may be interpreted as an integer or it may be interpreted as a floating-point number if it exceeds the C++ compiler's integer range. It is suggested that users adhere to the integer range of -32768 to 32767 regardless of their C++ compiler, in order to generate portable simulations which will run correctly on multiple platforms. To avoid writing simulations where certain numbers may be interpreted as integers on one computer and as floating-point numbers on another computer, add a decimal point and a zero (e.g. 40000.0) to integral numbers outside the range of -32768 to 32767 . This will guarantee that these numbers are always interpreted as floating-point numbers.

Floating-Point Numbers

Floating-point numbers consist of an optional plus or minus sign followed by one or more digits which are optionally followed by a period and one or more digits. Scientific (exponential) notation is supported. SGFramework implements floating-point numbers with double precision and with a guaranteed minimum range of -1.7×10^{-308} to 1.7×10^{308} with 15-digit precision. The syntax of floating-point numbers is as follows.

$[[+, -] digit + [[.] digit + [[E, e] [[+, -] digit +$

The numbers 1.0, -2.3, 1e1, 3.14159, $-2.0e - 3$, and $4.23e + 23$ are examples of floating-point numbers. Although numbers such as 5 and -10 are valid floating-point numbers as far as syntax is concerned, they will be treated as integers, since they are within every C++ compiler's integer range.

Integer and Floating-Point Expressions

Mathematical expressions may be divided into two categories: integer expressions and floating-point expressions. A mathematical expression which consists entirely of integers and operators will evaluate to an integer. Fractions are dropped in integer division. A mathematical expression which contains a floating-point number or a function will evaluate to a floating-point number. Consider the following examples.

9/2 evaluates to 4

9/2.0 evaluates to 4.5

$\log(100)$ evaluates to 2.0

$2 * 2 * 2 * 2$ evaluates to 16

The first expression evaluates to the integer 4, since the expression consists entirely of integers and the division operator. The second expression, however, evaluates to the floating-point number 4.5, since it contains a mixture of an integer and a floating-point number. The third expression evaluates to the floating-point number 2.0, since it contains the \log function. The fourth expression evaluates to the integer 16, since it consists entirely of integers and the multiplication operator.

In addition, integer and floating-point expressions may be subdivided into two further categories: constant expressions and variable expressions. Constant expressions are expressions that the SGFramework translator can evaluate to a constant at translation time. Constant expressions consist of numbers, constants, operators, internal functions and user-defined functions. Variable expressions are expressions that the SGFramework translator cannot evaluate to a constant at translation time. They contain at least one variable, array element, function argument, or external function. As a final note, integer expressions are always constant, since they cannot contain variables, array elements, function arguments, or external functions.

1.1.3 Strings

A string is a collection of one or more characters enclosed by double quotes. Every displayable ASCII character except the double quote (") character is a valid string character. Strings may not be broken over multiple lines in a specification file. The syntax of strings is as follows.

`["] character + ["]` where *character* is any printable ASCII character except for the double quote character.

Examples of strings are as follows.

"Hello,"

"Welcome to SGFramework."

"This is an example of a valid string."

1.1.4 Identifiers

Identifiers are used as constant, variable, array and user-defined function names. Certain identifiers are reserved as keywords. Since keywords have a special meaning, they should not be used as constant, variable, array and user-defined function names. Doing so will result in an error. Furthermore, SGFramework contains over 40 internal functions. The names of internal functions also should not be used as constant, variable, array and user-defined function names. Consult the internal function section for the names of the internal functions.

Naming Convention

Identifiers must begin with a letter, which is optionally followed by one or more letters or digits.

voltage, Electron, h0Le, cf4

are examples of valid identifiers, whereas

_voltage, elec_hole, 9lives, cf4+

are examples of invalid identifiers. Identifiers are not case sensitive unless the case-sensitive command switch is specified. Only the first seven characters of an identifier are guaranteed to be significant. Therefore the identifiers CH2OHCHOHCH2OH and CH2OHCHOHCH2NH2 may be identical and should not be used. The syntax of identifiers is as follows.

*letter letter , digit ** where *letter* is the set [A-Z,a-z].

Keywords

Keywords are not case sensitive. Keywords may not be used as identifiers, since the translator interprets them in special ways. Using a keyword in place of an identifier will usually result in a syntax error. SGFramework reserves the following identifiers as keywords.

'accuracy',
'algorithm',
'all',
'and',
'append',
'assign',
'begin',
'call',
'close',
'comment',

'const',
'coordinates',
'damping',
'diagonal',
'divisions',
'do',
'edge',
'else',
'end',
'exit',
'equ',
'file',
'fill',
'func',
'gausselim',
'goto',
'if',
'ilu',
'infinity',
'iterations',
'known',
'label',
'length',
'linear',
'linsol',
'logarithmic',
'main',
'maximum',
'minimum',
'newton',
'no',
'none',
'not',
'open',
'or',
'over',
'pcg',
'point',
'preconditioner',
'read',
'real',
'rectangles',
'refine',
'region',

Table: Operator Precedence

Operations	Associativity	Precedence
(unary) +, (unary) -, not	right-to-left	highest
*, /	left-to-right	
+, -	left-to-right	
>, <, >=, <=, ==, <>	left-to-right	
and, or, xor	left-to-right	lowest

'return',
 'set',
 'signedlog',
 'solve',
 'sor',
 'sort',
 'unknown',
 'var',
 'while',
 'write',
 'xor',
 'yes', and
 'zero'.

1.1.5 Operators

SGFramework supports the addition, subtraction, multiplication, division, negation and exponentiation operators. The precedence of these operators is shown in the table.

The unary plus, unary minus, addition, subtraction, multiplication, and division operators (+, -, *, /) behave as expected. The greater than, less than, greater than or equal to, less than or equal to, equal to, and not equal to operators (>, <, >=, <=, ==, <>) evaluate to 0 or 1 if both the left and right operands are integer expressions, or 0.0 or 1.0 if either the left or right operands are floating-point expressions. If the expression is true, unity is the result; while if the expression is false, zero is the result. The not, and, or and exclusive or (xor) operators likewise return unity if the expression is true and zero if the expression is false. However, these operators assume that nonzero operands are true. Hence the expression '5 and 3' evaluates to 1. Left-to-right associative operators with the same precedence are evaluated from left to right. Right-to-left associative operators with the same precedence are evaluated from right to left. Consider the following examples.

$4 + 5 * -6$ evaluates as $4 + (5 * (-6)) = 4 + -30 = -26$
 $3 - 4 - 7.0$ evaluates as $(3 - 4) - 7.0 = -1 - 7.0 = -8.0$
 $3 > 2 \text{ and } 4 > 9$ evaluates as $(3 > 2) \text{ and } (4 > 9) = 1 \text{ and } 0 = 0$
 $5 \text{ xor not } 4 + 2$ evaluates as $5 \text{ xor } ((\text{not } 4) + 2) = 5 \text{ xor } (0 + 2) = 0$

In the first example, the operations are performed in the following order: negation (unary minus), multiplication and addition. In the second example, the left subtraction operation is performed first followed by the right subtraction operation. Notice the result is a floating point number as the last operator is the subtraction of an integer and a floating-point number. In the next example, the greater than operators are evaluated before the and operator. The first greater than operation is true, hence unity is the result. The second greater than operation is false, hence zero is the result. The fourth example uses both boolean and mathematical operators. The not operator has highest precedence, so it is evaluated first. Since its operand is true (nonzero) it evaluates to zero (false). Next the addition operator is evaluated and finally the exclusive or operator.

1.1.6 Constants

A constant is a value that is fixed - it does not change during the simulation. Proper use of constants can make a specification file more readable and manageable.

Declaring Constants

To declare a constant one must type the keyword 'CONST' followed by an identifier, an equal sign, an expression which evaluates to a constant and a semicolon. Multiple constants may be declared in the same constant statement. The syntax of the constant statement is as follows.

CONST *identifier* '=' *const expr* | ',' ... *identifier* '=' *const expr* | ';' ;

Examples of constant statements are as follows:

```
CONST T = 300;           // temperature (K)
CONST Ks = 11.8;        // dielectric constant of Si
CONST k = 1.381e-23;    // Boltzmann's constant (J/K)
CONST q = 1.602e-19;    // charge of an electron (C)
CONST Vo = k*T/q;       // potential scaling (V)

CONST MIN = 2, MEAN = 8, MAX = 16;
CONST TRUE = 1, FALSE = 0;
CONST A = 20, B = sq(A), C = ln(B);
```

A constant expression is an expression that evaluates to a constant. All previously defined constants and user-defined functions as well as numbers, operators and internal functions may appear in constant expressions. Variables and array elements, on the other hand, cannot appear in constant expressions, since their values need not remain constant throughout a simulation. This is not to say that variables or array elements cannot be initialized to a value which does not change during a simulation. However, in general the values of variables and array elements may change during a simulation. Since the translator has no knowledge of which variables and array elements remain constant and which change, all variables and array

elements are excluded from constant expressions. See the following sections for more information concerning variables, arrays, user-defined functions and internal functions.

Using Constants

It is advantageous to use constants when the same value appears many times throughout a specification file. Consider the following specification file which solves Poisson's equation on a nine-by-nine square grid with uniform spacing. The electrostatic potential is held at zero on the boundary and 10 (volts) at the center.

```
var      V[9,9];
unknown V[1..7,1..7];
known    V[4,4];

equ V[i=1..7,j=1..7] ->
    V[i-1,j] + V[i,j-1] - 4.0*V[i,j] +
    V[i+1,j] + V[i-1,j] = 0.0;

begin main
    assign V[i=4,j=4] = 10.0;
    solve;
    write;
end
```

Suppose we decide that the simulation is not accurate enough. To remedy this problem, we wish to more than double the number of grid points. Let us choose a twenty-one by twenty-one grid. Since the dimensions of the array V are explicitly set to nine, we need to manually change all of the integer quantities in this simulation, i.e., the nines to twenty-ones, the sevens to nineteens, and the fours to tens. If we ran this simulation and decided we needed still more accuracy, we would have to repeat this process again. A better way to write this simulation is to define a constant which specifies the dimensions of the array. Consider the following simulation.

```
const DIM = 9;           // number of x and y grid points
const Vcenter = 10.0;    // voltage at center of grid (volts)

// define array V and make all array elements unknowns
// except for those at the center and on the boundary
var      V[DIM,DIM];
unknown V[1..DIM-2,1..DIM-2];
known    V[DIM/2,DIM/2];

// Poisson's equation
equ V[i=1..DIM-2,j=1..DIM-2] ->
    V[i-1,j] + V[i,j-1] - 4.0*V[i,j] +
```

```

V[i+1,j] + V[i-1,j] = 0.0;

// initialize center voltage, solve Poisson equation and
// write the results
begin main
  assign V[i=4,j=4] = Vcenter;
  solve;
  write;
end

```

Now if we ran this simulation and decided we needed more accuracy (suppose a twenty-one by twenty-one grid), we would only have to change the value of the constant DIM. Although the two specification files are equivalent in function, the second specification file is advantageous. It is much easier to change the number of grid points in the second simulation than the first. Furthermore, the lack of comments in the first specification file make its purpose more difficult to determine.

1.1.7 Variables and Arrays

Variables are blocks of memory that store floating-point numbers whose value may change during a simulation. Every variable must be declared before it can be used in a SGFramework specification file. Failure to declare a variable before using it will result in an error. Array variables, usually called arrays, are a group of variables that are referenced by a common identifier and one to three indices. The use of arrays can greatly simplify the writing of specification files by reducing redundancy.

Declaring Variables

It is easy to declare variables in SGFramework. To declare one or more variables, type the keyword 'VAR' followed by one or more identifiers, separated by commas. As with all SGFramework statements, the variable declaration statement must end with a semicolon. The syntax of the variable declaration statement is as follows.

```
VAR identifier | ',' ... identifier | ';' ;
```

Examples of variable declaration statements follow.

```

VAR voltage;
VAR ElectronConcentration, HoleConcentration;
VAR GasMileage, OdometerReading, TotalDistance;
VAR CF4,H2O,CaCl;
VAR A,B,C;

```

It should be emphasized that only the first seven characters of variable names are guaranteed to be significant. Therefore the following declaration statements may result in errors.

```

VAR ElectronLifetime, ElectronMobility;
VAR ElectrostaticPotential, ElectrostaticField;

```

Declaring Arrays

Declaring arrays is very similar to declaring variables. The only difference is that the array size must also be specified. To declare an array, type the keyword 'VAR', an identifier and the number of items to be stored, with the number surrounded by square brackets. Like variables, multiple arrays may be declared in the same statement with the arrays separated by commas. The statement must end with a semicolon. The syntax of the array declaration statement is as follows.

VAR *identifier* '[' *int expr* ']' | ',' ... *identifier* '[' *int expr* ']' | ','

Examples of array declaration statements are as follows:

```
VAR DaysOfTheWeek[7], MonthsOfTheYear[12];
```

```
VAR V[20], n[20], p[20];
```

In the above examples the size of the array was specified by an integer. However, any mathematical expression that evaluates to an integral constant is also acceptable. (See subsection 1.1.2 for more information on integral constant expressions.) Furthermore, multidimensional arrays (arrays which are referenced by more than one index) may be declared. The maximum number of dimensions an array may have is three. The syntax of the multidimensional array declaration statement is as follows.

VAR *identifier* '[' *int expr* | ',' ... *int expr* | ']' | ',' ... *identifier* '[' *int expr* | ',' ... *int expr* | ']' | ','

Variables, single-dimensional arrays and multidimensional arrays may all be declared in the same statement. The following examples demonstrate this point.

```
CONST DIM      = 9;
CONST HOLES    = 18;
CONST PLAYERS  = 15;
```

```
VAR V[DIM];
VAR Scores[PLAYERS, HOLES];
VAR Points[4*DIM-6, 3+10/4], Series, Strikes[PLAYERS];
```

The number of elements (size) in each array dimension must be greater than or equal to one. If the size of any dimension of an array evaluates to zero, a negative integer or a floating-point number, then an error will occur. As a final note, consider the first array declaration in the above examples. An array named V is declared with 9 members (since DIM is a constant equal to 9). The individual members of this array are referred to as V[0], V[1], V[2], ... V[7], V[8]. When referencing an array element the index may be between zero and the size of the array dimension minus one. For multidimensional arrays each index may be between zero and that dimension's size minus one. SGFramework follows the C++ language array indexing convention, which is different from those for languages such as Pascal and Fortran.

Specifying Unknown Variables

Typically not all of the variables or array elements declared in a specification file are unknown quantities whose value is determined by solving the equations. Consider the two-dimensional Poisson's equation file listed in Subsection 1.1.6. The center mesh point, array element $V[DIM/2, DIM/2]$, remains set to 10.0 volts throughout the simulation. Likewise the elements at mesh points on the perimeter of the mesh remain at zero volts throughout the simulation.

Because of this it is necessary to tell the SGFramework numerical algorithm modules which variables and array elements are the unknown quantities, for which a solution must be obtained. If this is not done, the equation headers are used to determine what is known and unknown - see 1.1.9. In order to declare variables or array elements as unknown, these variables and array elements must first be declared as variables using the VAR statement and then tagged as unknown variables using the UNKNOWN statement. Furthermore, 'unknown variables' may be untagged (converted back to known variables) by the KNOWN statement. The syntax of these statements is as follows.

```
UNKNOWN identifier [, ... identifier] ;
UNKNOWN identifier '[' range [, ... range] ']' [, ... identifier '[' range [, ...
range] ']' ;
KNOWN identifier [, ... identifier] ;
KNOWN identifier '[' range [, ... range] ']' [, ... identifier '[' range [, ...
range] ']' ;
```

Although it is not shown above, both variables and arrays may be present in the same UNKNOWN or KNOWN statement. Arrays require a range (or a collection of ranges) that specify which array elements should be tagged or untagged as known and unknown. The syntax of a range is as follows.

```
int expr [.. int expr [: int expr]]
all
```

The first integer expression is the range's starting value, the second integer expression is the range's ending value, and the third integer expression is the range's step value. All of the integer expressions must evaluate to an integer constant. If only the start value is specified, then range spans only that value. If the step value is omitted, then it defaults to plus or minus one. It is not allowable to specify a start value and a step value without the range's ending value, since such a range is meaningless. Examples of the KNOWN and UNKNOWN statements can be found in the example specification file given in Subsection 1.1.6 above and in the following examples. If the keyword 'ALL' is specified, then the range loops through all elements, i.e., its starting value is zero, its ending value is the size of the array dimension minus one, and its step is unity.

```
VAR Cell[3,3,3], Total, Time, x;
UNKNOWN Cell[0..2,1,0..2], Total, Time, x;
KNOWN Cell[1,0..2,1], Time;
```

Table: Internal Functions by Type

Group	Internal Function List
Trigonometric	sin, cos, tan, csc, sec, cot
Inverse Trigonometric	arcsin, arccos, arctan, arccsc, arcsec, arccot
Hyperbolic	sinh, cosh, tanh, csch, sech, coth
Inverse Hyperbolic	arcsinh, arccosh, arctanh, arccsch, arcsech, arccoth
Exponential and Logarithmic	exp, log, pow10, log, pow, sq, sqrt, inv
Miscellaneous	ave, bern, aux1, aux2, erf, nsdep, ngdep
Special	abs, sign, nonneg, step, min, max

After these KNOWN and UNKNOWN statements are processed, the following variables and array cells are unknown variables: *Total*, *x*, *Cell*[0, 1, 0], *Cell*[0, 1, 1], *Cell*[0, 1, 2], *Cell*[1, 1, 0], *Cell*[1, 1, 2], *Cell*[2, 1, 0], *Cell*[2, 1, 1] and *Cell*[2, 1, 2]. Note that subsequent KNOWN and UNKNOWN statements may override the results of previous statements.

1.1.8 Functions

SGFramework supports three types of functions: internal, user-defined, and external functions. Internal functions are built into the SGFramework language and need not be declared. User-defined functions are defined by the user in the specification file. External functions are declared in the specification but are implemented in some programming language, compiled, and then linked to the source code generated by SGFramework.

Internal Functions

SGFramework contains over forty internal functions which are available to every specification file for use. All SGFramework functions accept integral or floating-point expressions as arguments and return floating-point values. The internal functions can be divided into seven groups: trigonometric functions, inverse trigonometric functions, hyperbolic functions, inverse hyperbolic functions, exponential and logarithmic functions, miscellaneous functions and special functions whose derivatives are singular. The following table lists the internal functions according to their type; the table lists the syntax and a description of the internal functions in alphabetical order.

Since the max, min, nonneg, sign and step functions have singular derivatives and since SGFramework does not implement the partial derivatives of the nsdep and ngdep functions, it is recommended that these functions are not used in the constraint equations which are solved by SGFramework using a Newton method. (At a minimum, the user should be aware of the numerical issues resulting from the use of these functions in such equations.) If these functions are present in the constraint equations and have arguments containing unknown variables or array elements, then the Jacobian matrix will have singularities which may cause unde-

Table: Syntax/Description of Internal Functions

$\text{abs}(x)$	returns the absolute value of the argument x
$\text{arccos}(x)$	returns the inverse cosine of the argument x
$\text{arccosh}(x)$	returns the inverse hyperbolic cosine of the argument x
$\text{arccot}(x)$	returns the inverse cotangent of the argument x
$\text{arccoth}(x)$	returns the inverse hyperbolic cotangent of the argument x
$\text{arccsc}(x)$	returns the inverse cosecant of the argument x
$\text{arccsch}(x)$	returns the inverse hyperbolic cosecant of the argument x
$\text{arcsec}(x)$	returns the inverse secant of the argument x
$\text{arcsech}(x)$	returns the inverse hyperbolic secant of the argument x
$\text{arcsin}(x)$	returns the inverse sine of the argument x
$\text{arcsinh}(x)$	returns the inverse hyperbolic sine of the argument x
$\text{arctan}(x)$	returns the inverse tangent of the argument x
$\text{arctanh}(x)$	returns the inverse hyperbolic tangent of the argument x
$\text{aux1}(x)$	returns $x/\sinh(x)$
$\text{aux2}(x)$	returns $1/(1 + e^x)$
$\text{bern}(x)$	returns $x/(e^x - 1)$
$\text{ave}(x,y)$	returns average value of x and y
$\text{csc}(x)$	returns the cosecant of the argument x
$\text{csch}(x)$	returns the hyperbolic cosecant of the argument x
$\text{cos}(x)$	returns the cosine of the argument x
$\text{cosh}(x)$	returns the hyperbolic cosine of the argument x
$\text{cot}(x)$	returns the cotangent of the argument x
$\text{coth}(x)$	returns the hyperbolic cotangent of the argument x
$\text{erf}(x)$	returns $\frac{2}{\sqrt{\pi}} \int_0^x \exp(-s^2) ds$, the error function of the argument x
$\text{exp}(x)$	returns the exponential of the argument x , i.e. e to the x
$\text{inv}(x)$	returns the reciprocal of the argument x
$\text{ln}(x)$	returns the natural log of the argument x
$\text{log}(x)$	returns the log to base 10 of the argument x
$\text{max}(x,y)$	returns x if $x > y$, otherwise y is returned
$\text{min}(x,y)$	returns x if $x < y$, otherwise y is returned
$\text{ngdep}(x,y,W,a_x,a_y)$	if $\hat{x} = x - W/2$ is positive, returns $\exp(-a_x \hat{x}^2 - a_y y^2)$; otherwise returns 1.0
$\text{nonneg}(x)$	returns 1.0 if $x \geq 0.0$, otherwise 0.0 is returned
$\text{nsdep}(x,W,Dt)$	returns $0.5 \left[\text{erf} \left(\frac{W/2+x}{2\sqrt{Dt}} \right) + \text{erf} \left(\frac{W/2-x}{2\sqrt{Dt}} \right) \right]$
$\text{pow}(x,y)$	returns x to the power y , i.e. x^y
$\text{pow10}(x)$	returns 10 to the power x , i.e. 10^x
$\text{sec}(x)$	returns the secant of the argument x
$\text{sech}(x)$	returns the hyperbolic secant of the argument x
$\text{sign}(x)$	returns 1.0 if $x > 0.0$, returns 0.0 if $x = 0.0$ and -1.0 if $x < 0.0$
$\text{sin}(x)$	returns the sine of the argument x
$\text{sinh}(x)$	returns the hyperbolic sine of the argument x
$\text{sq}(x)$	returns the square of the argument x
$\text{sqrt}(x)$	returns the square root of the argument x
$\text{step}(x)$	returns 1.0 if $x \geq 0.0$, otherwise 0.0 is returned
$\text{tan}(x)$	returns the tangent of the argument x
$\text{tanh}(x)$	returns the hyperbolic tangent of the argument x

sirable behavior during the numerical solution of the equations. Examples of the use of internal functions can be found in the specification files listed in this manual. This applies to virtually all the semiconductor simulations, beginning with the simulation listed in Chapter ??, in the input file pn01.sg.

User-Defined Functions

In many simulations it is very convenient to declare and use application-specific functions. It is typical to implement parameter models as user-defined functions. For instance, parameter models for the electron and hole mobilities are implemented as user-defined functions in the simulations presented in this book. Once declared, user-defined functions may be used everywhere that internal functions are allowed. User-defined functions consist of a header and a body, as discussed below.

User-Defined Function Headers To declare a user-defined function header, one must type the keyword `FUNC` followed by an identifier, and a parenthesized list of function arguments. The identifier following the `FUNC` keyword is the user-defined function's name. The argument list consists of a comma-separated list of identifiers. Each identifier is the name of a function argument. Each argument may optionally be enclosed by a pair of less than '`<`' and greater than '`>`' signs.

`FUNC '(' [<] identifier [>] | ',' ... [<] identifier [>] | ')'`

Normally, each user-defined function is symbolically differentiated with respect to each function argument. Both the function and its partial derivatives are then coded in the output file. The partial derivatives are needed, since functions, in general, may be used in equations, which in turn are differentiated with respect to each unknown variable and array element. The equations are differentiated to obtain expressions that evaluate the elements of the Jacobian matrix. If, for some reason, the user does not wish to generate the *i*th partial derivative, then the user can enclose the *i*th argument between less than and greater than signs. There are at least two reasons why one might not want to generate a partial derivative of a user-defined function. First, one may know that the partial derivative always evaluates to zero. Second, one may not want to include the contribution of a partial derivative when evaluating the elements of the Jacobian matrix. For instance, in semiconductor simulations, the distance between mesh points is frequently a user-defined function argument. Since this distance is usually constant with respect to the simulation's unknown variables, there is no need to symbolically differentiate

the function with respect to this argument or to evaluate this partial derivative. Hence this argument is one which is often enclosed within less than and greater than signs.

User-Defined Function Bodies The body of a user-defined function consists of zero or more function statements, followed by a return statement. Two types of function statements are allowed: constant statements and assignment statements. The syntax of function constant statements is identical to the syntax of the constant statements described in Section 1.1.6. The expression which defines the constant statement must evaluate to a constant. Hence, all previously defined 'general'

constants and user-defined functions as well as numbers, operators, and internal functions may appear in constant expressions. Variables and array elements, on the other hand, cannot appear in a constant expression. In addition, the constant expression may contain the function constants which have been previously defined in the function in which the constant statement appears. One or more constants may be defined in a single function constant statement. The syntax of the function constant statement is as follows.

CONST *identifier* '=' *const expr* | ',' ... *identifier* '=' *const expr* | ';'

The only difference between general constants (constants declared outside of a function) and function constants is the scope of the constant. General constants may be used from their point of declaration to the end of the file. Function constants may be used only from their point of declaration to the end of the function in which they are defined. The names of function constants need not be unique throughout the file. The bodies of several user-defined functions may define the same constant. The value of the constant may also be different in any or all of the user-defined functions in which it is defined. Second, the name of a function constant may be the same as the name of a general constant. For instance, consider the following specification file.

```
const COUNT = 9;
var a,b,c;

func MyFunc(x,y)
    const TOTAL = COUNT + 3;
    const COUNT = TOTAL;
    return (x < y) * COUNT;

begin main
    assign a = COUNT, b = 10;
    assign c = MyFunc(a,b);
end
```

The first statement declares the COUNT constant whose value is 9. The second statement declares three variables. Next the function MyFunc is declared. MyFunc declares two function constants: TOTAL and COUNT. TOTAL is equal to COUNT plus 3, that is, 12. Note that COUNT, in this case, refers to the general constant COUNT whose value is 9. SGFramework does this because the function constant COUNT has not been declared yet. The second function statement defines a function constant named COUNT and assigns it the value of TOTAL, which is 12. MyFunc's return value is COUNT if x is less than y , or 0 if x is greater than or equal to y . Note that COUNT, in this case, refers to the function constant COUNT whose value is 12, not the general constant COUNT whose value is 9. Since functions return floating-point numbers, MyFunc actually returns the value 12.0 or 0.0. The integer constant is automatically converted to a floating-point number by the return statement. Lastly, let us look at the main procedure. The first statement

assigns the variable a the value of the general constant COUNT, which is 9, and assigns the variable b the value 10. The variable c is assigned the value of MyFunc whose arguments are a and b . Since the function arguments x and y correspond to variables a and b , and x is less than y , the variable c is assigned the value 12.0, which is the (floating-point) value of the function constant COUNT.

The second type of optional body statement is a function assignment statement. Function assignment statements declare temporary variables that may be used in the function in which they are declared. The syntax of the function assignment statement is the keyword 'ASSIGN' followed by an identifier that is the variable's name, an equal sign, a function assignment expression and a semicolon.

ASSIGN *identifier* '=' *func expr* ';' ... *identifier* '=' *func expr* ';' ;

The function assignment expression may contain constants, numbers, internal variables, previously declared user-defined and external functions, internal functions, the function arguments and mathematical operators. One or more function variables may be declared in a single function assignment statement by comma-separating the declarations and terminating the last declaration with a semicolon.

The function body must end with a return statement. The syntax of the return statement consists of the keyword RETURN followed by a return expression and a semicolon.

RETURN *func expr* ';' ;

The syntax of the return expression is identical to that of the function assignment expression. The value of the return expression is the value which is returned by the user-defined function. If the expression evaluates to an integral value, the value is automatically converted to a floating-point number.

Consider the three following user-defined functions.

```
func ave3(x, y, z)
  assign sum = x + y + z ;
  return sum / 3.0 ;
```

```
func grad(y1, y2, <h>)
  return (y2 - y1) / h ;
```

```
function AreaOfCircle(r)
  const PI = 3.141592654 ;
  return PI*sq(r) ;
```

The first function returns the average of its three arguments. It does this by first summing its three arguments and storing the result in a function variable. Then it returns one-third of the sum.

The second function returns the finite-difference approximation to the (one-dimensional) gradient. The arguments y_1 and y_2 will typically be the values of some scalar field at adjacent mesh points, and h is the distance between the mesh points. Since the location of the mesh points is often fixed, throughout the duration of the simulation, the derivative of the grad function with respect to the argument

h is not needed. To instruct the simulation not to evaluate this particular partial derivative, the h argument is enclosed by ' $<$ ' and ' $>$ '.

The last function returns the area of a circle with radius r . To do this the function defines the local (function) constant PI . It then returns PI times the square of the radius.

External Functions

In addition to internal and user-defined functions, SGFramework supports external functions. External functions can be used anywhere that internal or user-defined functions can be used, except in constant expressions. External functions are functions whose bodies are not defined in the specification file. Instead the body of an external function is implemented in a programming language (such as C, C++, Fortran or Pascal.) Consequently the user must compile and link external functions with the SGFramework-generated code and a numerical algorithm module (NAM). Since implementing these functions requires a thorough understanding of code and data structures generated by the SGFramework translator, this topic will not be discussed further here. External functions are scarcely used in the simulations presented in this book, and, when they are used, the source-code implementation is provided and is well-documented.

1.1.9 Constraint Equations

The constraint equations usually form a system of linear or nonlinear equations which a SGFramework numerical algorithm module solves to obtain the values of the unknown variables. If one is modeling a physical system, these equations constitute a mathematical description of physical laws (or approximations to those laws) as well as fixed relationships (constraints) between the system variables. For instance, if a system of masses and springs is modeled, application of Newton's and Hooke's laws provides the simulation's constraint equations. If the electrostatic interaction of a system of point charges is modeled, application of Coulomb's law provides the simulation's constraint equations. If the governing equations are partial differential equations, the constraint equations usually will be the corresponding discretized form of the PDEs. (We use the words 'constraint equations' here in a manner which is not entirely consistent with the notion of constraint equations as used in Lagrangian and Hamiltonian mechanics.)

The syntax of the constraint-equation statement is as follows.

EQU *header* ' $>$ ' *expr* = *expr* ' $<$ '

The use of mesh labels within constraint equations is described in Section 1.2, and especially in Subsection 1.3.1, after mesh labels have been introduced.

Constraint Equation Headers

The constraint-equation header consists either of a variable name or an array name followed by one to three loops enclosed in brackets. The syntax of the header is as

follows.

identifier identifier '[' *loop* | ',' ... *loop* | ']

The syntax of a loop is similar to that of a range with the addition of an index identifier. The syntax of a loop is as follows.

identifier '=' *range*

Loop indices are similar to variables with the following exceptions. First, loop indices store integer quantities, whereas variables store floating-point quantities. Second, the scope of a loop index is limited to the equation statement in which it appears. There are three implications of the second statement. First, the name of a loop index may be reused in subsequent equations. Second, if two or more loops appear in a header, then the indices must be unique. Third, indices may appear in the range expression of subsequent loops of the same statement but not in the range expressions of the loop in which the index is defined.

The equation headers serve two purposes. First, the headers may specify which variables and array elements are unknown. If KNOWN and UNKNOWN statements do not appear in the specification file, SGFramework uses the equation headers to determine which variables are unknown. Each variable and array element specified by the array header is assumed to be unknown in the absence of KNOWN and UNKNOWN statements. However, if any KNOWN or UNKNOWN statements appear in the file, all declared variables and array elements are assumed to be known unless explicitly tagged as unknown by an UNKNOWN statement. For more information about KNOWN and UNKNOWN statements, refer to Subsection 1.1.7.

The main purpose of equation headers is to associate equations with unknown variables. If the header is a variable, then the equation is associated with that variable. Consequently, the variable which appears in the header should not be explicitly tagged as a known variable via a KNOWN statement. Furthermore, if a KNOWN or UNKNOWN statement appears in the specification file, header variables should explicitly be tagged as unknown. If an equation is associated with a known variable, then SGFramework will issue a warning that the equation is useless.

Frequently, one wishes to solve a system of discretized partial differential equations when simulating a physical system. Often the resulting discretized equations can be represented by a couple of template equations, i.e., equations which have the same form but different array indices. Consider the SGFramework specification file of Subsection 1.1.6. This specification file numerically solves Poisson's equation on a nine by nine rectangular mesh. The electrostatic potential, V , is held constant at the center of the mesh and its perimeter. Consequently, the V array has 81 elements, 48 of which are tagged as unknown variables. Since there are 48 unknowns in this specification file, there must be 48 constraint equations as well. These equations are as follows.

$$\begin{aligned} V[0,1] + V[1,0] - 4 * V[1,1] + V[2,1] + V[1,2] &= 0 \\ V[0,2] + V[1,1] - 4 * V[1,2] + V[2,2] + V[1,3] &= 0 \\ V[0,3] + V[1,2] - 4 * V[1,3] + V[2,3] + V[1,4] &= 0 \end{aligned}$$

$$V[6,7] + V[7,6] - 4 * V[7,7] + V[8,7] + V[7,8] = 0$$

Since all of these constraint equations have the same form and differ only in the values of their array indices, they may be represented by the following template (indexed) constraint equation.

$$V[i-1,j] + V[i,j-1] - 4 * V[i,j] + V[i+1,j] + V[i,j+1] = 0$$

Both the indices i and j loop through the values one through seven, i.e., $i = 1..7$ and $j = 1..7$. Consequently the header for this constraint equation would be $V[i=1..7,j=1..7]$. Since the electrostatic potential is known at the center, there is no equation associated with the array element $V[4,4]$.

When SGFramework processes equation statements, it associates that equation with either a variable or one or more array elements. An equation is tagged 'useless' if SGFramework could not associate the equation with one or more unknown variables and/or array elements. This warning is usually generated if (1) the variable or at least one array element with which one wishes to associate the equation is not specified as unknown or (2) the variable or array elements are already associated with a previous equation. It is important to recognize that SGFramework associates a variable or array element with the first equation it encounters that specifies that variable or array element in its header. Consider the following example.

```
var x[10];
unknown x[all];

equ x[j=0..7] -> x[j] = 5.0;
equ x[j=8..9] -> x[j] = 3.0;
equ x[j=7..8] -> x[j] = 4.0;
```

The last equation is 'useless', since the equation associated with array element $x[7]$ is the first equation and the equation associated with array element $x[8]$ is the second.

Constraint Equation Expressions

Constraint-equation expressions are mathematical expressions consisting of previously defined constants, variables, array elements, external functions, internal functions and user-defined functions as well as operators and the index variables of the loops that are specified in the constraint equation header. Consider the following system of equations.

$$\begin{aligned} x - y - z &= 0 \\ x - y + z &= 2 \\ x + y + z &= 6 \end{aligned}$$

The above equation can easily be implemented as follows:

```
var x, y, z;  
equ x -> x - y - z = 0 ;  
equ y -> x - y + z = 2 ;  
equ z -> x + y + z = 6 ;
```

It worth noting that in the above system of equations, any of the equations could be associated with any of the variables. In general, this is not true. One cannot associate a variable or array element with an equation in which the variable or array element does not appear in the equation's expression.

1.1.10 Procedures

Procedures are needed for several reasons in a simulation. Usually it is necessary to initialize variables and array elements prior to solving the specified equations in order to determine the values of the unknown variables and/or array elements. (By default, all variables and/or array elements are initialized to zero.) In addition it is often desirable to solve the equations for different boundary conditions and to output simulation results to data files.

Procedures are the mechanism by which users can initialize their variables and/or array elements, tell the numerical algorithm modules to solve their specified equations and write their simulation results to data files. Procedures are each specified by a unique name and contain one or more procedural statements. The syntax for procedures is:

```
begin identifier  
  procedural statement ';' ;  
  procedural statement ';' ;  
  ...  
end
```

Procedures start with a header that consists of the keyword 'begin', followed by an identifier which serves as the procedure's name. Procedures terminate with the keyword end. Between the header and terminator are one or more procedural statements. In addition, SGFramework supports external procedures. External procedures are directly coded in a programming language such as C/C++, Fortran or Pascal. External procedures are specified by generating a procedure with no body (i.e., a procedure with a header directly followed by the terminating keyword 'end'.)

Every simulation must have at least one procedure, the main procedure. The syntax of the main procedure is identical to that of the generic procedure described above, except for the header. The main procedure's header consists of the keyword 'begin' followed by the keyword 'main'. Upon execution of a SGFramework simulation, the simulation initializes its internal data structures and numerical algorithm module. After this start-up code is complete, program control is turned over to the main procedure. Each statement in the main procedure is executed, branching to and returning from subroutines (other procedures) as necessary. When the end

of the main procedure is reached, the simulation calls its clean-up code and the simulation terminates.

Between the procedure's header and terminator are one or more procedural statements. There are six types of procedural statements: assignment statements, single-word statements, file input/output statements, subroutine execution statements, conditional statements and looping statements. Each of these statement types will be discussed in detail.

Assignment Statements

Assignment statements are used initialize and/or modify the values of variables and array elements. By default the values of all simulation variables and array elements are set to zero upon invoking the simulation. This default may be overridden by means of assignment statements for some or all of the variables and array elements. Assignment statements may also modify the values of variables and array elements in the middle of simulation to simulate time varying boundary conditions. Finally, assignment statements may calculate quantities based upon other variables and elements. For instance, assignment statements are used in the semiconductor simulations

1. To initialize variables and array elements,
2. To modify the values of boundary conditions such as contact voltages which provide the interface between regions where the physical equations are solved, and
3. To calculate quantities such as terminal currents in terms of variables and array elements.

The difference between assignment statements in procedures and those in functions is as follows. Assignment statements in functions both declare and initialize variables that are visible only in the function where they are declared. Assignment statements in procedures do not declare new variables or array elements. They are strictly used to modify the values of existing variables and array elements that are visible to the entire program, i.e. those variables and array elements which have been declared in VARIABLE statements. The syntax of ASSIGNMENT statements is as follows.

```
ASSIGN identifier = expr ';'

```

```
ASSIGN identifier '[' loop | ',' ... loop | ']' = expr ';'

```

Procedural assignment statements begin with the keyword 'assign' followed by an identifier, which is the name either of a previously declared variable or of a previously declared array element. If the identifier is the name of an array element, then it must be followed by a comma-separated list of loops enclosed in brackets. The loops specify which array elements are included in the assignment statement. See Section 1.1.9 for more information concerning loops. Whether the identifier is the name of a variable or of an array, the statement is completed by an equal

sign, an expression and a semicolon. The expression may be any mathematically valid formula composed of previously declared constants, variables, array elements, functions as well as numbers, internal functions, and operators (as well as loop indices, if applicable).

Examples of assignment statements can be seen throughout this book in numerous specification files. For example, in the ‘Game of Life’ specification file that is listed in the introduction, the following assignment statements were used:

```
assign t=0.0 ;

assign AO[i=is,j=js+1] = 1 ;      // launch stop light
assign AO[i=is-1..is+1,j=js] = 1 ; //

assign AO[i=ig,j=jg+2] = 1 ;      // launch glider
assign AO[i=ig+1,j=jg+1] = 1 ;    //

assign AO[i=ig-1..ig+1,j=jg] = 1 ; //

assign A[i=all,j=all] = AO[i,j] ;

assign COUNT[i=1..LX-1,j=1..LY-1] =
  AO[i-1,j-1] + AO[i-1,j] + AO[i-1,j+1] +
  AO[i,j-1]   + AO[i,j+1]   +
  AO[i+1,j-1] + AO[i+1,j] + AO[i+1,j+1];

assign A[i=1..LX-1,j=1..LY-1] =
  (COUNT[i,j]==3) or (COUNT[i,j]+AO[i,j]==3);
  assign AO[i=all,j=all] = A[i,j] ;
assign t = t+dt ;
```

These statements are largely self-explanatory. Some assignment statements used above are for a single variable, some refer to a single element of an array, and some apply to a specified range of elements of an array. (To see how this fits into the overall input file, see Section ??.)

Single-Word Statements

There are three procedural statements that consist of single keywords: solve, write, and exit. The syntax is the keyword followed by a semicolon.

```
solve ';'
write ';'
exit ';'

```

The solve statement causes the simulation to invoke the appropriate numerical analysis algorithm to solve the specified equations to obtain the values of the unknown variables and array elements. The equations are solved, starting by using the current values of all of the simulation variables and array elements. The values of the unknown variables and array elements are modified as a consequence of

invoking the solve statement.

The write statement dumps a snapshot of all the variables and array elements to a binary data file, referred to as the result file. If a write statement is not explicitly present in the simulation, an implicit write statement is appended after the last statement in the main procedure. It is possible to override the behavior of the write statement to dump a snapshot of selected variables and array elements by writing code to customize the SGFramework default behavior. Data in the SGFramework data files may be extracted and viewed using certain SGFramework tools such as 'extract' and 'tripplot'. The extract command is discussed in Section 1.4.8. The tripplot command is discussed in the section on Graphical Output, 1.4.10.

The exit statement causes the simulation to call its clean-up code and then terminates the simulation.

Input and Output

In many circumstances, it is convenient to initialize variables and/or array elements from numbers stored in data files rather than computing their values from a formula. It may also be expedient to write the values of certain variables and/or array elements to ASCII data files rather than dumping the values of all variables and/or arrays to a binary result file. Towards this end, SGFramework provides several commands to accomplish the input and output needs of most users.

Data Files SGFramework data files are ASCII files which can contain three types of data: numerical data (floating-point numbers), labels and comments. The most prevalent type of data in SGFramework data files is numerical data. Numerical data is always treated as floating-point numbers, since this data either will initialize variables and/or array elements or was generated by writing the values of variables and/or array elements. Numerical data need not have any special format. It is only required that each number be separated by whitespace (spaces, tabs, new lines, etc.).

Labels mark a particular place in the data file in the same fashion as tabs mark a book. When reading numerical data from an input file, one can instruct the simulation to search for a particular label and read data starting from that point in the input file. This feature is useful when several simulations use the same data file to initialize their variables and/or array elements. Common data can be stored at the beginning of the input file and simulation-specific data can be tagged with a label and stored at the end of the data file.

Data files may also have comments. Like SGFramework mesh and equation specification files, comments begin with two slash characters '/' and end at the end of the line. Comments are ignored when reading numerical data and searching for labels. Comments serve no other purpose than to annotate the data file for human readability.

Opening Data Files In order to read data from or write data to an ASCII data file, it is necessary first to open the file. The syntax for the open file statement is

as follows. The file open command begins with the keyword OPEN followed by a string and a file mode. The command is terminated by a semicolon.

OPEN ["] *file name* ["] *file mode* ';'

The string following the 'OPEN' keyword specifies the full name of the file, i.e., path, name, and extension. The file mode is one of the following keywords: 'READ', 'WRITE', and 'APPEND'. The READ file mode opens the data file for input of data from the file. The WRITE and APPEND file modes open the data file for output to the file. The differences between WRITE and APPEND are apparent only when trying to open an existing data file. WRITE will erase and overwrite the contents of the data file, whereas APPEND does what its name implies: it appends the new data to the end of the existing data.

File Pointer When a file is opened for data input from the file, a pointer, referred to as the file pointer, is positioned at the beginning of the file. When the simulation reads numerical data from the file, it reads the first number that is located at or following the file pointer. After it reads the value, the file pointer is advanced past the data which was just read. Similarly, when the simulation is searching for a label in a data file, it starts its search at the file pointer. If the search key is found, the file pointer is updated to the position which immediately follows the label. If the label is not found, the file pointer will point to the end of the data file. To locate a search key which is located prior to the file pointer's current location, the file must first be closed and then reopened. Reopening the data file causes the file pointer to be reset to the beginning of the file.

When a file is opened for data output to the file, the file pointer is located at either the beginning or the end of the file, depending upon the mode in which the file was opened. If the data file is opened with the WRITE mode, then the file pointer is located at the beginning of the file. If the file is opened with the APPEND mode, then the file pointer is located at the end of the file (unless, of course, the file does not exist, in which case the beginning and the end of the file are the same). Whenever numerical data, comments, or labels are written to the output file, the information is written at the current location of the file pointer. The file pointer is then updated to point to the end of the information that was written.

Reading from a Data File Once a file is open for input from the file, i.e., opened with the READ file mode, numerical data from the file can be read into simulation variables and array elements via the FILE READ command. The syntax for this command is as follows. The FILE READ command begins with the keywords 'FILE' and 'READ' and is followed by a comma-separated list of identifiers. The command is terminated by a semicolon. The identifiers in the list are the names of variables and arrays into which the numerical data in the data file will be read. If an identifier refers to the name of a simulation variable, only one number is read from the data file. If, however, an identifier refers to the name of a simulation array, then several numbers are read from the data file and stored sequentially in the elements of the array. It is not possible to read data into a select range of array elements. Data is read from the current position of the file pointer, and the file pointer is updated

upon execution of this command.

```
FILE READ identifier |',' ... identifier |','
```

Writing to a Data File Once a file is open for output to the file, i.e., opened with the WRITE or APPEND file mode, the values of simulation variables and array elements may be written to the data file via the FILE WRITE. The syntax for this command is similar to the syntax of the FILE READ command. The FILE WRITE command begins with the keywords 'FILE' and 'WRITE' and is followed by a comma-separated list of identifiers. The command is terminated by a semicolon. The identifiers in the list are the names of variables and arrays whose values will be written to the data file. If an identifier refers to the name of a simulation variable, its value will be written to the data file. If, however, an identifier refers to the name of a simulation array, then the values of each of its array elements will be sequentially written to the data file. It is not possible to write data from a selected range of array elements. Data is written to the current position of the file pointer, and the file pointer is updated upon execution of this command.

```
FILE WRITE identifier |',' ... identifier |','
```

Closing a Data File Once data has either been read from or written to a data file, the file needs to be closed before another file can be opened. It is a recommended practice to always close a data file regardless of whether another file will be opened. The syntax of the close file statement is very simple. The command starts with the keyword 'CLOSE' and is terminated by a semicolon.

```
CLOSE ','
```

Subroutine Execution Statements

As stated in the introduction to the procedure subsection, every SGFramework simulation must have a main procedure. When a SGFramework simulation is invoked, an initialization procedure is first executed. After the simulation has initialized its internal data structures, it then transfers program control to the main procedure.

Many simulations have additional procedures. Procedures provide a convenient mechanism of grouping a series of related statements into a unit, which makes the equation specification file more modular. A procedure can call another procedure via a call statement. Typically, the statements in a procedure are executed sequentially. When a call statement is encountered, program control is transferred to the procedure that is called. Once all of the statements in the called procedure have been executed, program control is transferred back to the calling procedure. The syntax of the call statement is as follows.

```
CALL identifier ','
```

Call statements begin the keyword 'CALL', followed by an identifier and a semicolon. The identifier is the name of a procedure which is to be executed.

Consider the following example.

```
var A, B, C;
```

```
begin MyProc1
  assign C = A + B;
end

begin MyProc2
  assign A = A + 1.0;
  assign B = B - 2.0;
  call MyProc1;
end

begin main
  assign A = 3.0;
  assign B = 7.0;
  call MyProc2;
  assign C = 2.0 * C;
end
```

After this simulation is invoked and it completes its initialization procedure, program control is transferred to the main procedure. The statements in the main procedure are sequentially executed. Hence, the first two assignment statements are executed which initialize variables A and B to 3.0 and 7.0 respectively. The next statement calls the procedure MyProc2. Hence the statements in procedure MyProc2 begin to execute sequentially. The first two assignment statements in MyProc2 reassign the values of variables A and B to 4.0 and 5.0 respectively. The next statement in procedure MyProc2 calls the procedure MyProc1. Consequently, the procedure MyProc1 is executed which assigns the variable C the value of 9.0 (the sum of variables A and B.) Since procedure MyProc1 only has one statement, program control returns to the procedure that called procedure MyProc1, i.e. procedure MyProc2. Since procedure MyProc2 does not have any more statements after its call statement, program control returns to the procedure which called MyProc2, i.e. the main procedure. Execution in the main procedure continues after the call statement. Hence, the assignment statement which reassigns variable C to twice its value is executed. At the end of this simulation, the variables A, B, and C have the values of 4.0, 5.0, and 18.0 respectively.

Conditional Statements

All of the statements discussed thus far, with the exception of procedure calls, execute sequentially. In other words, the statements are executed in the order in which they appear in the main procedure. Often it is desirable to execute different statements depending upon the result of some computation or condition. Conditional statements provide such a mechanism. SGFramework supports IF-THEN and IF-THEN-ELSE conditional statements. By combining several conditional statements, it is possible to construct ELSE-IF conditional statements as well. The syntax for these statements is as follows.

```

IF '(' condition ') THEN statement
IF '(' condition ') THEN statement ELSE statement .

```

IF-THEN statements begin with the keyword 'IF' and are followed by a parenthesized condition. After the condition, the keyword 'THEN' is specified followed by one SGFramework procedural statement. Conditions are simply mathematical expressions. The condition is treated as being TRUE if it evaluates to a nonzero value and FALSE if it evaluates to zero. The procedural statement is executed only if the condition is TRUE. IF-THEN-ELSE statements have a similar syntax. Instead of the statement terminating with a procedural statement, the keyword 'ELSE' and another procedural statement are appended. In IF-THEN-ELSE statements, if the condition is TRUE, the first procedural statement is executed (the one after the THEN keyword). If the condition is FALSE, the second procedural statement is executed (the one after the ELSE keyword).

It is often desirable to conditionally execute several procedural statements rather than one procedural statement. To accommodate this feature, SGFramework allows several statements to be treated as one by enclosing the statements between the keywords 'BEGIN' and 'END'. All statements discussed in this section thus far are valid procedural statements. Furthermore, conditional statements and branching statements (which will be discussed next) are also valid procedural statements. For instance, consider the following example.

```

VAR a, b, c;

IF      ( a > b ) THEN
  ASSIGN c = +1.0;
ELSE IF ( a < b ) THEN
  ASSIGN c = -1.0;
ELSE
  ASSIGN c = 0.0;

```

This example compares the values of variables a and b and stores the result in variable c . If a is greater than b , then the result is positive unity. If a is less than b , then the result is negative unity. If a and b are equal, then the result is zero. Notice that in this example, an IF-THEN-ELSE statement is used as the procedural statement of another IF-THEN-ELSE statement. If the condition ' a is greater than b ' is true, then the assignment statement ' c is set to positive unity' is executed. If the condition ' a is greater than b ' is not true, then the statement that begins with 'if a is less than b ' is executed.

A slight variation of the use of conditional statements within conditional statements is illustrated in the next example.

```

VAR a, b, c;

ASSIGN c = 0.0;
IF      (a > 0.0) THEN

```

```

BEGIN
  IF      (b > 0.0) THEN ASSIGN c = +1.0;
  ELSE IF (b < 0.0) THEN ASSIGN c = -1.0;
END
ELSE IF (a < 0.0) THEN
  BEGIN
    IF      (b > 0.0) THEN ASSIGN c = -1.0;
    ELSE IF (b < 0.0) THEN ASSIGN c = +1.0;
  END

```

These statements check the sign of variables a and b and store the result in variable c . If either a or b is zero, then the result is zero. If a and b are both positive or both negative numbers, then the result is positive unity; otherwise, the result is negative unity. In this case, we wanted the first IF-THEN-ELSE statement to execute if a is greater than zero and the second IF-THEN-ELSE statement to execute if a is less than zero. Therefore, these conditional statements had to be enclosed by the 'BEGIN' and 'END' keywords. Otherwise, it would not be clear whether the first 'ELSE' keyword should be associated with the first or second 'IF' keyword.

Looping Statements

The last type of procedural statements are looping statements. Looping statements allow any procedural statement (or a group of procedural statements if they are enclosed by the BEGIN and END keywords) to be executed repeatedly as long as some condition remains TRUE. SGFramework supports two types of looping statements: WHILE statements and DO-WHILE statements. The syntax of these commands is as follows.

```

WHILE '(' condition ')' statement
DO statement WHILE '(' condition ')' ';'

```

WHILE statements begin with the keyword 'WHILE' and are followed by a parenthesized condition and a procedural statement. As long as the condition remains TRUE, then the procedural statement will be executed over and over. If the condition becomes FALSE, then the procedural statement is skipped and the procedural statement which immediately follows the entire WHILE statement is executed.

DO-WHILE statements begin with the keyword 'DO' and are followed by a procedural statement, the keyword 'WHILE', a parenthesized condition and a semi-colon. Like WHILE statements, the procedural statement will be executed over and over as long as the condition remains TRUE. The difference between these looping statements is apparent when the condition is FALSE prior to executing the looping statement. In the case of a WHILE statement, the procedural statement will never be executed if the conditional is initially FALSE. However, with the DO-WHILE statement, the procedural statement will be executed once. Hence the procedural statement of a WHILE statement may be executed zero or more times and the

procedural statement of a DO-WHILE statement may be executed one or more times.

In most cases, the procedural statement of a loop will consist of several procedural statements enclosed by the 'BEGIN' and 'END' keywords. At least one of the enclosed statements should affect the value of the condition; otherwise, the loop will never end. SGFramework omits FOR loops, since these may be implemented with a *while* statement. The following example will loop through the statements between the 'BEGIN' and 'END' keywords ten times. Note, however, that the loop variable *i* is a floating-point number and not an integer.

```
ASSIGN i = 1.0;
WHILE (i < 10.1) BEGIN
    statement
    statement
    ...
    statement
    ASSIGN i = i + 1.0;
END
```

1.1.11 Numerical Algorithm Parameters

Equation specification files may also specify several parameters. Parameters are divided into three categories: nonlinear solution algorithm parameters, linear solution algorithm parameters, and mesh scaling parameters. Parameters are specified via SET statements. To understand the parameters specified in this example, it is important to understand how a nonlinear system of algebraic equations is solved. It is recommended that the reader review section ?? and the sections which follow.

Nonlinear Solution Algorithm Parameters

There are three nonlinear solution algorithm parameters. These parameters specify the maximum number of Newton iterations, minimum accuracy with which to solve the simulation's equations, and the maximum amount of damping that can be applied to the Newton update vector. This subsection assumes the reader is familiar with the Newton algorithm.

The parameter statement that specifies the maximum number of Newton iterations begins the keywords 'SET', 'NEWTON', and 'ITERATIONS'. These keywords are followed by an equal sign, an integer, and a semi-colon. The integer specifies the maximum number of Newton iterations. The numerical algorithm module will continue to iterate, as long as this maximum number of iterations is not exceeded and the simulation equations have not been solved to the minimum specified accuracy (and no error has been detected). The syntax of this parameter statement is as follows.

```
SET NEWTON ITERATIONS '=' integer ';'
```

The parameter statement that specifies the minimum accuracy with which to

solve the simulation's equations begins with the keywords 'SET', 'NEWTON', and 'ACCURACY'. Following these keywords is an equal sign, a floating-point (real) number, and a semicolon. The floating-point number specifies the minimum accuracy. The syntax of this parameter statement is as follows.

```
SET NEWTON ACCURACY '=' real ';'
```

The parameter statement that specifies the maximum amount by which to damp the Newton update vector begins with the keywords 'SET', 'NEWTON', and 'DAMPING'. These keywords are followed by an equal sign, an integer, and a semicolon. The maximum amount of damping is given by the expression 2.0^{-i} where i will be set equal to the integer mentioned in the previous sentence. The syntax of this parameter statement is as follows.

```
SET NEWTON DAMPING '=' integer ';'
```

Linear Solution Algorithm Parameters

There are seven linear solution algorithm parameters. These parameters specify the linear solution algorithm, the maximum matrix fill level, the conjugate gradient preconditioner, the maximum number of linear solution iterations, the minimum accuracy of the linear solution algorithm, the value of the overrelaxation parameter, and whether to zero the Newton update vector prior to executing the linear solution algorithm.

The parameter statement that specifies the linear solution algorithm begins with the keywords 'SET', 'LINSOL', and 'ALGORITHM'. These keywords are followed by an equal sign, the keyword 'SOR', 'PCG', or 'GE', and a semicolon. If the linear solution algorithm parameter is set to 'SOR', then the successive overrelaxation algorithm is used. If this parameter is set to 'PCG', then the preconditioned conjugate gradient algorithm is used. Finally, if this parameter is set to 'GE', then the Gaussian elimination algorithm is used. The syntax of this parameter statement is as follows.

```
SET LINSOL ALGORITHM '=' SOR ';'  
SET LINSOL ALGORITHM '=' PCG ';'  
SET LINSOL ALGORITHM '=' GE ';'
```

The parameter statement that specifies the maximum levels of matrix fill begins with the keywords 'SET', 'LINSOL', and 'FILL'. These keywords are followed by an equal sign, an integer, and a semicolon. The integer specifies the maximum matrix fill level. This parameter is only used when the preconditioned conjugate gradient with an ILU preconditioner or Gaussian elimination algorithm is used as the linear solution solver. The syntax of this parameter statement is as follows.

```
SET LINSOL FILL '=' integer ';'  
SET LINSOL FILL '=' INFINITY ';'
```

The parameter statement that specifies the conjugate gradient preconditioner begins with the keywords 'SET', 'LINSOL', and 'PRECONDITIONER'. These keywords are followed by an equal sign, the keyword 'NONE', 'DIAGONAL', or 'ILU', and a semicolon. If this parameter is set to 'NONE', then no preconditioner is used.

If this parameter is set to 'DIAGONAL', then the preconditioner is the matrix whose diagonal elements are $d_{ii} = \partial F_i / \partial x_i$ (where F_i represents the i^{th} equation and x_i represents the associated variable) and whose off diagonal elements are zero. If this parameter is set to 'ILU', then the preconditioner is the incomplete LU factored Jacobian matrix. This parameter is only used when the preconditioned conjugate gradient is the linear solution solver. The syntax of this parameter statement is as follows.

```
SET LINSOL PRECONDITIONER '=' NONE ';'
SET LINSOL PRECONDITIONER '=' DIAGONAL ';'
SET LINSOL PRECONDITIONER '=' ILU ';'
```

The parameter statement that specifies the maximum number of linear solution iterations begins with the keywords 'SET', 'LINSOL', and 'ITERATIONS'. These keywords are followed by an equal sign, an integer, and a semi colon. The integer specifies the maximum number of linear solution iterations. This parameter is only used when the successive overrelaxation or preconditioned conjugate gradient algorithm is the linear solution solver. The syntax of this parameter statement is as follows.

```
SET LINSOL ITERATIONS '=' integer ';'
```

The parameter statement that specifies the minimum linear solution accuracy begins with the keywords 'SET', 'LINSOL', and 'ACCURACY'. These keywords are followed by an equal sign, a floating-point (real) number, and a semicolon. The floating-point number is the minimum linear solution accuracy. This parameter is only used when the successive overrelaxation or preconditioned conjugate gradient algorithm is the linear solution solver. The syntax of this parameter statement is as follows.

```
SET LINSOL ACCURACY '=' real ';'
```

The parameter statement that specifies the value of the overrelaxation parameter begins with the keywords 'SET', 'LINSOL', and 'OVER'. These keywords are followed by an equal sign, a floating-point (real) number, and a semicolon. This parameter is only used when the successive overrelaxation algorithm is the linear solution solver. The syntax of this parameter statement is as follows.

```
SET LINSOL OVER '=' real ';'
```

The parameter that specifies whether the Newton vector update is zeroed prior to executing the linear solution algorithm begins with keywords 'SET', 'LINSOL', and 'ZERO'. The keywords are followed by an equal sign, the keyword 'YES' or 'NO', and a semicolon. If the value of this parameter is 'YES', then the Newton update vector is zeroed prior to calling the linear solution algorithm. If the value of this parameter is 'NO', then the Newton update vector is not zeroed prior to calling the linear solution algorithm. This parameter is only used when the successive overrelaxation or preconditioned conjugate gradient is the linear solution algorithm. Note, the Newton update vector serves as an initial guess for the indirect linear solution solvers. The syntax of this statement is as follows.

```
SET LINSOL ZERO '=' YES ';'
SET LINSOL ZERO '=' NO ';'
```

Mesh Scaling Parameters

There is only one mesh scaling parameter. This parameter sets the mesh distance scaling. The mesh distance scaling parameter begins with the keywords ‘SET’ and ‘DIST’. These keywords are followed by an equal sign, a constant expression, and a semicolon. The constant expression evaluates to the value of the mesh distance scaling. The syntax of this parameter statement is as follows.

```
SET DISTANCE '=' const expr ';' ;
```

This concludes the discussion of the syntax and grammar of equation specification files. Following is a discussion of the syntax and grammar of mesh specification files.

1.2 The Syntax and Grammar of the Mesh Specification File

SGFramework provides the ability to automatically generate and refine hybrid rectangular/triangular two-dimensional meshes based on the specification contained in an input file. The algorithms and associated data structures are discussed in Chapter ???. This section focuses on describing the syntax and grammar of the mesh specification file. This section is divided into several subsections: An Overview of Mesh Specification Files 1.2.1; Comments, Numbers, Identifiers and Constants 1.2.2; Coordinates 1.2.3; Points 1.2.4; Edges 1.2.5; Regions 1.2.6; Labels; 1.2.7; Refinement Statements 1.2.8; Mesh Parameters 1.2.9; and the Element Refinement Criteria 1.2.10.

1.2.1 An Overview of the Mesh Specification File

SGFramework mesh specification files are usually divided into two sections: the mesh skeleton and the mesh refinement criteria. The mesh skeleton contains the minimum amount of information necessary to accurately describe the mesh. The mesh skeleton consists of a hierarchical declaration of points, edges and regions. SGFramework performs several data consistency checks and is designed so that additional checks may be implemented easily. The mesh refinement criteria specify the minimum and maximum spacing between nodes, the minimum and maximum numbers of refinement divisions and the conditions under which to refine (divide) triangles and/or rectangles. The naming of all of the mesh quantities is important, since it is the mechanism which links the mesh specification and equation specification files. The equation specification files were previously referred to as the specification files.

A very simple mesh skeleton file ‘sq.sk’ was given in Section ???. The discussion there would serve as a simple introduction to this topic. As a more complete example, consider the MOSFET mesh specification file listed in Section ???. The file is duplicated below, for convenience.

```
// mos.sk
// mesh constants
```

[illegible]

```
// define points
point pA = ((WDEV-WOX)/2, DOX), pG = (WDEV-WCONT, 0.0);
point pB = ((WDEV+WOX)/2, DOX), pH = (WDEV, 0.0);
point pC = (0.0, 0.0), pI = ((WDEV-WOX)/2, -DRECT);
point pD = (WCONT, 0.0), pJ = ((WDEV+WOX)/2, -DRECT);
point pE = ((WDEV-WOX)/2, 0.0), pK = (0.0, -DDEV);
point pF = ((WDEV+WOX)/2, 0.0), pL = (WDEV, -DDEV);
```

```
// define edges
edge eAB = GATE    [pA, pB] (WOX/20, 0.0);
edge eDE = NOFLUX  [pE, pD] (1.0e-7, 0.5);
edge eEF = SISIO2  [pE, pF] (WOX/20, 0.0);
edge eFG = NOFLUX  [pF, pG] (1.0e-7, 0.5);
edge eIJ =          [pI, pJ] (WOX/20, 0.0);
edge eCD = DRAIN   [pC, pD] (WCONT/8, 0.0);
edge eAE = NOFLUX  [pE, pA] (1.0e-7, 0.5);
edge eGH = SOURCE  [pG, pH] (WCONT/8, 0.0);
edge eBF = NOFLUX  [pF, pB] (1.0e-7, 0.5);
edge eCK = NOFLUX  [pC, pK] (WCONT/8, 0.5);
edge eEI =          [pE, pI] (1.0e-7, 0.5);
edge eHL = NOFLUX  [pH, pL] (WCONT/8, 0.5);
edge eFJ =          [pF, pJ] (1.0e-7, 0.5);
edge eKL = SUB      [pK, pL] (WDEV/5, 0.0);
```

```
//define regions
```

```

region r1 = SiO2 {eAE, eEF, eBF, eAB} RECTANGLES;
region r2 = Si    {eEI, eIJ, eFJ, eEF} RECTANGLES;
region r3 = Si    {eCK, eKL, eHL, eGH, eFG, eFJ, eIJ, eEI, eDE, eCD};

// define coordinate labels
coordinates x, y;

// physical constants and properties of Si and SiO2
const T      = 300.0;           // operating temperature
const e      = 1.602e-19;       // electron charge (C)
const kb     = 1.381e-23;       // Boltzmann's constant (J/K)
const e0     = 8.854e-14;       // permittivity of vacuum (F/cm)
const eSi    = 11.8;           // dielectric constant of Si
const eSiO2  = 3.9;            // dielectric constant of SiO2

// doping constants
const NS = 1.0e16;              // substrate doping (cm-3)
const NC = 1.0e19;              // contact doping (cm-3)
const WDIFF = (WDEV-WOX)/2;     // diffusion width (cm)
const DDIFF = 0.25e-4;          // diffusion depth (cm)
const DT     = 1.0e-11;         // diffusion coef. * time (cm2)

// doping profile
refine C (SignedLog, 3.0) = (y <= 0.0) * { -NS +
  (NC+NS) * nsdep(x, 2*WDIFF,DT) * nsdep(y,2*DDIFF,DT) +
  (NC+NS) * nsdep(WDEV-x,2*WDIFF,DT) * nsdep(y,2*DDIFF,DT) };

// set min/max edge spacing and min/max refinement levels
set minimum length = sqrt(e0*eSi*(kb*T/e)/e/abs(C));
set maximum length = 1.0;
set minimum divisions = 0;
set maximum divisions = 20;

```

1.2.2 Comments, Numbers, Identifiers and Constants

Equation and mesh specification files have similarities in their syntax and grammar. Both of these specification files allow comments to be placed anywhere in the source file. Numbers, identifiers, and constants are also common between equation and mesh specification files.

Comments in mesh specification files are ignored by the mesh specification file parser. Comments start with the `//` characters and terminate at the end of the line. It is a recommended practice to use comments to document and annotate mesh specification files. Comments are used liberally in the MOSFET mesh specification file. For example, a series of comments towards the beginning of the example file provide a schematic of the MOSFET mesh. Furthermore, comments are used to explain the meaning of the constants declared in the file.

Mesh specification files also support integer and floating-point numbers. Integer

and floating-point expressions are also supported with one exception. Since mesh specification files do not have variables, all mathematical expressions evaluate to a constant. For more information about numbers, refer to Subsection 1.1.2.

Identifiers are used in mesh specification files as the names of constants, points, edges, regions, labels, coordinates, and refinement criteria parameters. As with equation specification files, certain identifiers are reserved as keywords. For more details about identifiers, refer to Subsection 1.1.4.

Finally, constants are supported in mesh specification files. The syntax of constants in mesh specification files is identical to the syntax of constants in equation specification files. The proper use of constants can make a mesh specification more readable and manageable. It is good practice to define geometrical quantities such as the device width and depth as constants as is done in the sample mesh specification file. Defining these quantities as constants allows users to easily modify the device geometry by simply changing the value of the constant. For more information about constants, refer to Subsection 1.1.6.

1.2.3 Coordinates

In order to access the coordinate values of the mesh points, one must first name the coordinates using a coordinate statement. Unlike any other mesh specification file statement, only one coordinate statement is allowed in a mesh specification file. The syntax of the coordinate statement is as follows.

`COORDINATES identifier ',' identifier ';'`

See Section 1.3.1 for details of how the coordinates of the mesh points are passed to the equation specification file.

In the MOSFET example, the coordinates are labeled 'x' and 'y', since the MOSFET geometry is specified in the Cartesian coordinate system. If the mesh specification file does not contain a coordinate statement, SGFramework labels the coordinates 'x' and 'y'.

1.2.4 Points

The basic building blocks of mesh specification files are points. Edges are defined by points, and regions are defined by edges. A point is defined by two coordinates. The coordinates are enclosed by parentheses. Each point is given a name which is an identifier. In the example given here, point names consist of the lowercase letter 'p' followed by an uppercase letter. For instance, the first point is named pA and the last point is named pL. In general, the names of objects must begin with a letter and may be followed by one or more letters and/or digits. For more information about naming points, refer to the identifier Subsection 1.1.4. The syntax of the point definition is as follows.

`POINT identifier '=' '(' const expr ',' const expr ')' '|' ',' ... identifier '=' '(' const expr ',' const expr ')' '|' ';'`

1.2.5 Edges

Edges in SGFramework mesh specification files may either be straight line segments or circular arcs. A straight line-segment is defined by its two endpoints. A circular arc is defined by its two endpoints and a third point on the arc. In each case the points defining the edge are enclosed by brackets. The edges in the example MOSFET mesh specification file are all straight line-segment edges. Each edge is given a name. In this example, edge names consist of the letter ‘e’ followed by two letters which indicate the points at the ends of the edges. This naming, however, is not dictated by the language. In general, the name of edges may be any legal identifier (see Subsection 1.1.4).

In addition, an edge may be given a label. For example, in the MOSFET example file, edge eAB is labeled GATE and edge eCD is labeled DRAIN. Edges without labels are considered internal edges. Points on an internal edge may be adjusted to improve the quality of the mesh, whereas points on a labeled edge may not be nudged. A detailed explanation of labels will be presented in Subsection 1.2.7.

The edge definition may also include an initial node spacing and a grade. The grid-generation program will use these parameters to control the insertion of points on the edge. If these parameters are omitted, the grid-generation program will compute appropriate values for them. For example, edge eAE is a line-segment that connects points pE and pA. Edge eAE has an initial node spacing of 10 angstroms (10^{-7} cm) and a grade of 0.50. This declaration causes the grid generation program to insert a point on the edge that is 10 angstroms from point pE. The next point will be spaced 15 angstroms from the previous point or 25 angstroms from point pE, so that the distance between the first and second nodes is 50 percent larger than the distance between the first point and point pE.

The syntax of the edge definition is

```
EDGE identifier '=' | '[' identifier ',' identifier | '[' identifier | ']' | '('
const expr ',' const expr ')' | ',' ... identifier '=' | identifier | '[' identifier ',' identifier
| ',' identifier | ']' | '(' const expr ',' const expr ')' || ';'

```

where the two or three identifiers in the square brackets define the endpoints, and, in the case of a curved circular edge, a third point on the edge. (The third point on the curve is the middle identifier). Curved edges should not exceed a quarter circle.

1.2.6 Regions

Regions are defined by a list of three or more edges. The list is enclosed between braces. The list of edges must form a simple closed curve. Regions include both the curve and its interior. The interiors of regions cannot overlap; however, regions may share boundaries (edges). Each region has a name. However, unlike edges where labels are optional, regions must be given a label. In this example, region names consist of the letter ‘r’ followed by a number. In general, region names may be any valid identifier (see Subsection 1.1.4). The edges must be listed in counterclockwise order in the definition of the region.

By default, regions are divided into triangular elements. Rectangular regions may be divided into rectangular elements if the opposite sides of the region have the same initial spacing and grade. The edges of rectangular regions must be parallel to the coordinate axes. There is one more constraint on rectangular regions. The opposite edges must 'point' in the same direction. For example, consider horizontal edges of a rectangular region. If the points are ordered from left to right on one of the edges, the points on the other edge must follow the same ordering. The user may specify rectangular elements by inserting the keyword 'RECTANGLES' at the end of the region statement.

The syntax of the region definition is

```
REGION identifier '=' identifier '{' identifier | ',' ... identifier | '}' [RECTANGLES]
| ',' ... identifier '=' identifier '{' identifier | ',' ... identifier | '}' [RECTANGLES] | ';'
where the identifiers in the brackets form a list of edges.
```

1.2.7 Labels

Labels provide links between the mesh specification and equation specification files. The distinction between names and labels is that names must be unique whereas the same label can be used to identify several edges and/or regions. For example, consider the second and third regions. These regions have different names ('r2' and 'r3') but share the same label ('SI'). Equation specification files refer to edges and regions by their labels (such as 'GATE' and 'DRAIN') while the mesh specification refers to points, edges and regions by their names (such as 'eAB' and 'eCD'). Labels refer to a collection of points. For instance, the label 'GATE' refers to all of the points on the edge 'eAB' whereas the label 'SI' refers to all of the points in the regions 'r1' and 'r2' and on their boundaries. Interfacing mesh and equation specification files with labels will be discussed in more detail in Subsection 1.3.2.

1.2.8 Refinement Statements

As mentioned in the overview, mesh specification files consist of two parts: a mesh skeleton and mesh refinement criteria. The mesh skeleton consists of a hierarchy of points, edges, and regions. From this information, the SGFramework mesh generation program constructs an initial grid. Often, this grid needs to be refined by dividing the triangular and/or rectangular elements in certain subdomains. The refinement criteria are specified by refinement functions.

The syntax of refinement functions is as follows.

```
REFINE identifier '(' identifier ',' const expr ')' '=' expr ';'
```

Refinement functions consist of a name, two refinement parameters and a body. The function name is declared after the keyword 'refine'. The refinement parameters are enclosed by parentheses and follow the function's name. The first refinement parameter is the refinement measure (which may be LINEAR, LOG or SLOG) and the second is the refinement distance. The body is the expression which follows the equal sign and may be a function of position. In this MOSFET example, the body of the refinement statement is the device's doping profile. The refinement

measure may be linear, log or signed log, as defined by equations (1.1), (1.2) and (1.3) respectively.

$$M_{\text{lin}}(x) = x \quad (1.1)$$

$$M_{\text{log}}(x) = \log(x) \quad (1.2)$$

$$M_{\text{slog}}(x) = \text{sign}(x) * \log(1.0 + |x|) \quad (1.3)$$

In order to determine whether a triangular or rectangular element should be refined, the mesh refinement program will evaluate the refinement functions at each of the element's vertices. If the difference in measure between any two vertices exceeds the refinement distance, the element is refined. For example, consider the MOSFET example. The refinement measure is signed log and the refinement distance is 4. In order to determine if a triangular element with vertices (x_1, y_1) , (x_2, y_2) and (x_3, y_3) should be refined, the mesh refinement program would first evaluate the mesh refinement function C at each vertex ($C_i = C(x_i, y_i)$ for $i = 1, 2, 3$). It would then check the measure between the vertices ($M_{12} = |M_{\text{slog}}(C_1) - M_{\text{slog}}(C_2)|$, $M_{23} = |M_{\text{slog}}(C_2) - M_{\text{slog}}(C_3)|$ and $M_{13} = |M_{\text{slog}}(C_1) - M_{\text{slog}}(C_3)|$). If $M_{12} > 4$, $M_{23} > 4$ or $M_{13} > 4$, then the triangle would be refined. Multiple mesh refinement functions may be declared. The aforementioned procedure is performed using each refinement function. The results of these tests are logically OR'ed together. Thus an element is refined if any one of the tests signifies that it should be refined.

1.2.9 Mesh Parameters

Mesh specification files may also specify several parameters such as the minimum and maximum number of divisions and the minimum and maximum lengths of edges. Each triangular or rectangular element in the initial grid is assigned a division level of zero. If an element is refined, then the resulting elements are assigned the division level of their parent plus one. Thus elements formed from the division of a level-zero element would have division level one and elements formed from the division of a level one element would have division level two. The minimum and maximum division parameters set the minimum and maximum number of divisions through which an element may be refined. The syntax of the statements that set these parameters is as follows.

```
SET MINIMUM DIVISIONS '=' const expr ';';
```

```
SET MAXIMUM DIVISIONS '=' const expr ';';
```

The minimum and maximum edge lengths may be functions of position. Furthermore, the maximum edge length may be specified independently in both coordinates. The minimum edge length in the MOSFET example is set to the Debye length. The maximum edge length is set to one micron in this example. The syntax of these parameters is as follows.

```
SET MINIMUM LENGTH '=' expr ';';
```

```
SET MAXIMUM LENGTH '=' expr ';';
```

```
SET MAXIMUM identifier LENGTH '=' expr ';';
```

Note that the identifier in the last syntax statement is the name of a coordinate label. Furthermore, the expressions which define the minimum and maximum lengths may be functions of position.

1.2.10 Element Refinement Criteria

The following rules are used in the order shown to determine whether an element should be refined. Once a decision has been reached, the remaining rules are not invoked for that element. An element can be refined only once on each pass over the mesh.

1. If two or more of the element's edges have been divided by the refinement of adjacent elements, then the element is divided.
2. If an element's division level is less than the minimum division level specified in the mesh input file, then the element is divided.
3. If the element's division level is greater than the maximum division level specified in the mesh input file, then the element is not divided.
4. If any of the lengths of the element's edges is larger than the maximum edge length specified in the mesh input file, then the element is divided.
5. If all of the lengths of the element's edges are smaller than the minimum edge length specified in the mesh input file, then the element is not divided.
6. If the criteria based on the refinement functions are not satisfied as discussed above, then the element is divided.

1.3 Interfacing the Equation and the Mesh Specification Files

In order to implement a finite-difference scheme, one needs to know how the mesh nodes are connected. In other words, each node must know who are its neighbors. This problem is trivial with rectangular meshes, since the simulation variables may be stored in multidimensional arrays which reflect the mesh geometry. Since, in general, the nodes of irregular meshes are not stored in any predictable order, it is not convenient to store the simulation variables in multidimensional arrays. With irregular meshes, the simulation variables are usually stored in one-dimensional arrays. In addition, one usually explicitly stores the mesh connectivity (lists of neighboring nodes for each node).

This section discusses the SGFramework language constructs that are used to interface the mesh and equation specification files. Specifically, this section describes the equation specification file extensions, such as mesh connectivity functions, that allow a user to write a simulation using an irregular mesh. This section is divided into five subsections: Importing an Irregular Mesh 1.3.1, Using Labels in Equation Specification Files 1.3.2, Mesh Connectivity Functions 1.3.3 and Mesh

Geometry Functions 1.3.4, Mesh Summation Functions 1.3.5, and Precomputed Functions 1.3.6.

1.3.1 Importing an Irregular Mesh

In order to write a simulation using an irregular mesh, the equation specification file must first import the mesh via the mesh statement. The syntax of the mesh statement is as follows.

```
MESH string ‘;
```

The mesh statement consists of the keyword ‘mesh’ followed by a string and a semicolon. The string is the name of the mesh to import.

Importing a mesh does several things. First of all, it imports all of the constants in the mesh specification file from which the mesh was generated. In other words, it allows the author of equation specification files to use the constants that are defined in the mesh specification file of meshes they import. For example, the example MOSFET mesh specification file declares the constants WDEV and DDEV. These constants define the width and depth of the MOSFET respectively. Any equation specification file that imports the mesh generated from the MOSFET mesh specification file can use constants WDEV and DDEV in the statements that follow the mesh statement.

Secondly, importing a mesh implicitly declares three additional constants: NODES, EDGES, and ELEMENTS. These constants define the number of nodes, edges, and elements (both triangular and rectangular) that are present in the imported mesh. These constants are often used to declare the number of elements in an array. For instance, suppose we wanted to write a simulation that computes the electrostatic potential of a MOSFET whose geometry and doping profile is defined by the example MOSFET mesh specification file. To do so, we would first import the MOSFET mesh using the mesh statement. We would then declare an array which would store the electrostatic potential at each node on the mesh. To do this, we need to know how many nodes the mesh contains. Thus, we use the NODES constant as shown in the following statements.

```
mesh "mosfet.msh";
var V[NODES];      // array V stores the electrostatic potential
```

Thirdly, importing a mesh imports all of the labels of the mesh specification file from which the mesh was created. Labels represent a collection or list of nodes. For instance, importing the mesh generated from the example MOSFET, would import the labels GATE, NOFLUX, SISIO2, DRAIN, SOURCE, SUB, SI02, and SI. The label GATE represents a list of nodes which are on the edge labeled GATE. The label SI02 represents a list of nodes which are in and on the boundary of the region labeled SI02. The use of labels in equation specification files is discussed in Subsection 1.3.2.

Finally, importing a mesh imports some arrays. All of these imported arrays are one-dimensional and the number of elements in each array is equal to the number

of mesh nodes. At least two arrays will be imported. The names of these two arrays will be the names given to the coordinate labels in the mesh specification file via the coordinate statement. The values of the elements of these arrays will be the positions of the mesh nodes. Furthermore, an additional array will be imported for each refinement function present in the imported mesh's specification file. The names of these arrays will be the names given to the refinement functions. The values of the elements of these arrays will be the bodies of the refinement functions evaluated at each mesh point. For example, the example MOSFET mesh specification file labels its coordinates 'x' and 'y'. In addition, this file defines one refinement function named 'C' whose body computes the doping profile. When meshes that are generated from this mesh specification file are imported, three arrays will be imported whose names are 'x', 'y', 'C'. The values of the elements of these arrays will be the x position, y position, and dopant concentration at each node in the mesh.

One word of caution needs to be given. Authors of equation specification files that import meshes should not declare constants, variables, functions, etc. whose names are identical to the names of the imported constants, labels, or arrays. If one does do this, a symbol redeclaration error will be generated.

1.3.2 Using Labels in Equation Specification Files

Several equation specification file statements such as known and unknown statements, equation statements, and assignment statements require a range or a loop when used in conjunction with arrays. The ranges and loops specify a group of indices that specify to which array elements the statements apply. (see 1.1.7 and 1.1.9).

For instance, consider the example MOSFET equation specification file. Suppose we want to determine the electrostatic potential throughout the device. As our first step, we import the mesh and declare an array to store the value of the electrostatic potential at each node (see Subsection 1.3.1). As our next step, we declare all the elements of the electrostatic potential array unknown except for the nodes on the gate, source, and drain contacts. As our final step, we assign the values of the elements of the electrostatic potential array at the gate, source, and drain contacts equal to the constants V_{gate} , V_{source} , and V_{drain} . The following example shows the statements which accomplish these steps.

```
mesh "mosfet.msh";

const Vgate = 2.0;
const Vsource = 0.0;
const Vdrain = 0.0;

var V[NODES];      // array V stores the electrostatic potential
unknown V[all];
known V[GATE], V[SOURCE], V[DRAIN];
```

```

begin main
  assign V[i=GATE] = Vgate;
  assign V[i=SOURCE] = Vsource;
  assign V[i=DRAIN] = Vdrain;
end

```

The above example illustrates the use of labels in equation specification files. Notice that labels, which are defined in the mesh specification file, are used as ranges. This is possible because ranges represent a group of nodes. One may ask, ‘Why not use a traditional range which specifies a starting index, an ending index, and a step value?’ The answer is three-fold. First, since we do not know which nodes reside on the gate, source, and drain contacts, we cannot define a range by specifying a starting index, an ending index, and a step value. Second, even if we did know which nodes reside on these contacts, in general, we still could not specify the starting index, ending index, and a step value because the nodes on any edge or region are not guaranteed to be sequential. Third, suppose we did know which nodes reside on the contacts and they could be represented by a range that is specified by a starting index, an ending index, and a loop. We still would not want to use this approach, since we would have to change the ranges’ starting indices, ending indices and step values every time we modified the mesh.

Upon importing an irregular mesh, simulation authors may use labels as ranges in any equation specification statement that requires a range (or loop, since a loop contains a range). The syntax of a range is extended as follows.

```

int expr |.. int expr |: int expr ||
all
identifier

```

Hence, ranges may be used in known and unknown statements, equation headers, and assignment statement headers. Ranges provide a convenient way to specify a group of indices. Furthermore, they allow users to change certain characteristics of the mesh without having to change the equation specification file that imports the mesh. For instance, we could change the depth of the example MOSFET mesh specification file by changing the value of the WDEV constant. Since this change does not modify the mesh labels, no change would be required in a properly written equation specification file that imports the MOSFET mesh.

1.3.3 Mesh Connectivity Functions

As mentioned in the introduction to this section, the nodes of irregular meshes, in general, are not stored in any easily predictable order. Therefore, it is necessary to explicitly store the mesh connectivity. SGFramework provides two functions to access this information: node and edge. The syntax of these functions is as follows.

```

NODE '(' int expr ',' int expr ')'
EDGE '(' int expr ',' int expr ')'

```

The mesh connectivity functions begin with the keyword ‘NODE’ or ‘EDGE’

and are followed by a left parenthesis, an integer expression, a comma, another integer expression, and a right parenthesis. The first integer expression specifies a mesh node while the second integer expression specifies a neighbor index. Examples of mesh connectivity functions are `node(i,j)` and `edge(i,j)`. The function `node(i,j)` returns an index to the j th neighbor of the i th node. The function `edge(i,j)` returns an index to the edge which connects the i th node to its j th neighbor. This function does not return the edge which connects the i th and j th node. Note that these functions are exceptional, in that they return an integer number as opposed to a floating-point quantity. Hence, node and edge functions may appear in the index expressions of arrays. For instance, the expression

```
V[i] - V[node(i,j)]
```

computes the difference between the values of array elements that correspond to the i th node and its j th neighbor.

1.3.4 Mesh Geometry Functions

It is often desirable to know geometrical quantities such as the distance between the i th node and its j th neighbor. One could compute this quantity via the following expression.

```
sqrt(sq(x[i]-x[node(i,j)])+sq(y[i]-y[node(i,j)]))
```

This, however, requires two differences to be computed and three function calls. Since the distances between a node and its neighbors are often used in simulations, these values have been precomputed and can be accessed via the edge length function.

SGFramework provides several functions that access certain precomputed geometrical quantities of irregular meshes: the edge length function (`elen`), the integration edge length function (`ilen`), the integration area function (`area`), and the partial integration area function (`area`). The syntax of each of these functions is as follows.

```
ELEN '(' int expr ',' int expr ')'
ILEN '(' int expr ',' int expr ')'
AREA '(' int expr ')'
AREA '(' int expr ',' int expr ')'
```

The first integer expression in each function evaluates to a node. The second integer expression (if it exists) evaluates to a neighbor index. To understand these functions, consider Figure 1.1. This figure illustrates the quantities that the mesh geometry function returns. Consider the node that is represented by the solid circle. An integration 'box' is formed by connecting the perpendicular bisectors of the edges that connect the node to its neighbors. The perpendicular bisectors are referred to as integration edges. The `elen(i,j)` function returns the length of the edge that connects the i th node and its j th neighbor (not the length of the edge that connects the i th

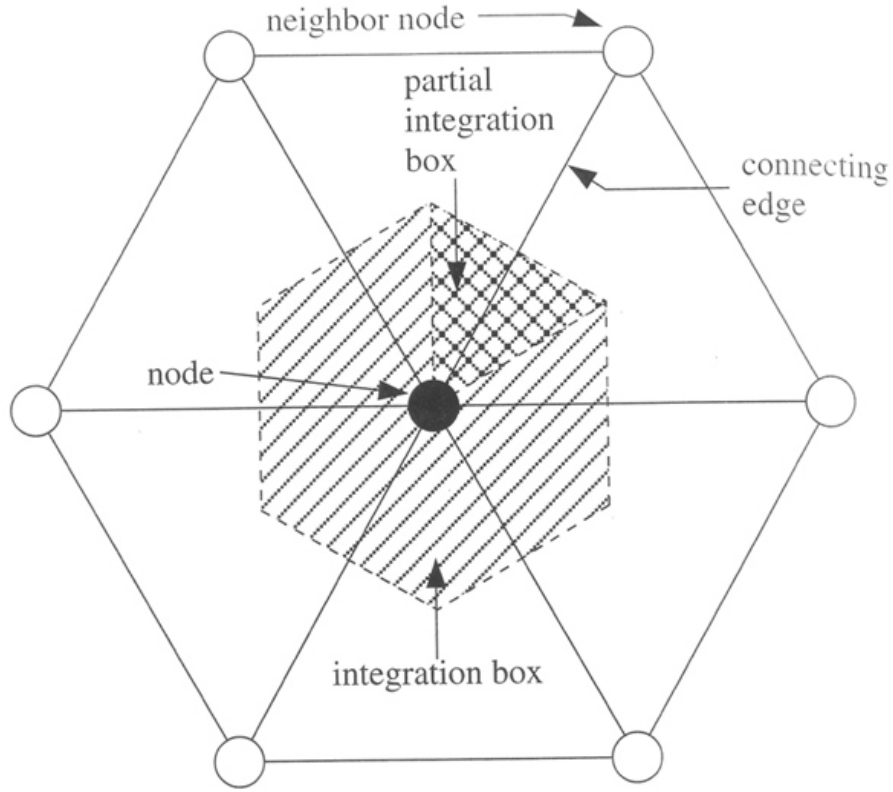


Figure 1.1. A schematic diagram illustrating the edge length, integration edge length, the integration area, and the partial integration area.

and j th nodes). The $ilen(i,j)$ function returns the length of integration edge that perpendicular bisects the edge that connects the i th node and its j th neighbor. The $area(i)$ function returns the area of the integration box that contains the i th node. Finally, the $area(i,j)$ function returns the partial area of the integration box that is formed by the triangle whose vertices consist of the i th node and the endpoints of the integration edge discussed in the description of the $ilen(i,j)$ function.¹

¹The figures in this chapter are reprinted from Computer Physics Communications, Vol. 93, "Strategies for mesh-handling and model specification within a highly flexible simulation framework", 179-211, 1995, with permission from Elsevier Science - NL, Amsterdam, the Netherlands.

1.3.5 Mesh Summation Functions

It is often desirable to evaluate and sum an expression over a group of nodes. The SGFramework language provides two summation functions: the label summation function (lsum) and the node summation function (nsum). This subsection is divided into three parts. The first part discusses the label summation function. The second part describes the node summation function. The third part describes which summation statements may be nested inside other summation statements.

Label Summation Function

Labels represent a group of nodes. The label summation function loops over a group of nodes represented by a label, evaluates an expression at each node, and sums the results. The syntax of the label summation function is as follows.

```
LSUM '(' identifier ',' identifier ',' expr ')'
```

Label summation functions start with the keyword 'LSUM' followed by three comma-separated parameters enclosed by parentheses. The first parameter is an identifier. This identifier is the name of an index whose value is set to the index of the summation's current node. The second parameter is another identifier. This identifier is the name of a label that specifies the group of nodes the summation loops over. The last parameter is a mathematical expression. This expression is evaluated and summed at each node of the specified label. Usually this expression 'is a function of' the specified index. By 'is a function of', we mean that the specified index appears in the mathematical expression.

Node Summation Function

Whereas the label summation functions loops over a group of nodes represented by a label, the node summation function loops over the neighbors of a specified node. The syntax of the node summation function is as follows.

```
NSUM '(' int expr ',' identifier ',' identifier ',' expr ')'
```

```
NSUM '(' int expr ',' identifier ',' ALL ',' expr ')'
```

Node summation functions start with the keyword 'NSUM' followed by four comma-separated parameters enclosed by parentheses. The first parameter is an integer expression that evaluates to the node whose neighbors are looped over. The second parameter is an identifier. This identifier is the name of an index whose value is set to the index of the summation's current node. The third parameter is either an identifier which is the name of a label or the keyword 'ALL'. If this parameter is a label, then it filters the nodes over which the summation loops. Only those neighbors that are part of the group represented by the specified label are included in the summation. If this parameter is the keyword 'ALL', then all of the neighbors are included in the summation. The last parameter is a mathematical expression. This expression is evaluated and summed at each allowed neighbor. Usually this expression 'is a function of' the node given by the first parameter and the index specified by the second parameter. By 'is a function of', we mean that these indices appear in the mathematical expression.

Nesting Summation Functions

The last parameter of both summation functions is a mathematical expression. In general, one is not allowed to nest summation functions. In other words, one can not include another summation function in a summation function's expression. The only exception to this rule is that node summation functions may be nested in label summation functions.

To illustrate summation functions, consider a simulation which imports a mesh generated from the example MOSFET mesh specification file. Suppose this simulation solves for the MOSFET's electrostatic potential (V), electron concentration (n), and hole concentration (p) at each node of the mesh. The simulation declares user-defined functions which return the electron and hole current densities. The following code computes the MOSFET drain's current per unit length.

```
mesh "mosfet.msh";

func Jn(n1,n2,V1,V2,<h>)
...

func Jp(p1,p2,V1,V2,<h>)
...

var V[NODES], n[NODES], p[NODES];
...

begin main
  call Initialize;
  solve;
  assign I = lsum(i,DRAIN,nsum(i,j,SI,ilen(i,j)*
    {Jn(n[i],n[node(i,j)],V[i],V[node(i,j)],elen(i,j))+
      Jp(p[i],p[node(i,j)],V[i],V[node(i,j)],elen(i,j))}));
end
```

1.3.6 Precomputed Functions

Since a significant portion of the simulation time may be spent evaluating the elements of the Jacobian matrix, it is desirable to make this process as efficient as possible. One manner in which this may be accomplished is by eliminating redundant calculations by storing intermediate values in memory. To this end, the SGFramework language provides a mechanism that allows user-defined functions (and their partial derivatives with respect to their arguments) to be precomputed and stored in arrays for quick retrieval. Precomputation of user-defined functions is performed prior to constructing the Jacobian matrix or the function vector. It should be noted that function precomputation is only supported in simulations where SGFramework meshes are imported. This subsection is divided into two parts. The first part describes how to specify precomputed functions and the second part describes how to use precomputed functions.

Specifying Precomputed Functions

After a user-defined function has been declared, it may be tagged for precomputation via precompute statements. Not all user-defined functions may be precomputed. Precomputation is limited to user-defined functions that compute a value at a mesh node or a mesh edge. Specifying precomputation for each of the types of user-defined function will be discussed individually.

Specifying Node Precomputed Functions Precompute statements for user-defined functions that compute a value at a node begin with the keyword 'PRECOMPUTE' and are followed by an 'at' (@) symbol, the keyword 'NODE', an identifier, an arrow, a function, and a semicolon. The identifier is the name of a node index which may be used by specified user-defined function's arguments. The syntax of these precompute statements is as follows.

PRECOMPUTE '@' NODE *identifier* '->' *function* ';'

Examples of precompute statements for user-defined functions that compute a value at a node are given below.

```
var V[NODES], n[NODES], p[NODES], C[NODES], tn[NODES], tp[NODES];

func MUn(<N>,<T>,pn)
...
return MU1*[1.025/[1+pow(X/1.68,1.43)]-0.025]/u0;

func R(n,p,<tn1>,<tp1>)
...
return Rsrh+Raug;

precompute @ NODE i -> MUn(C[i],T,n[i]*p[i]);
precompute @ NODE i -> R(n[i],p[i],tn[i],tp[i]);
```

The first precompute statement specifies that the user-defined function MUn should be evaluated at node i with arguments N, T, and pn equal to C[i], T, and n[i]*p[i] respectively. The last precompute statement specifies that the user-defined function R should be evaluated at node i with arguments n, p, tn1, and tp1 equal to n[i], p[i], tn[i], and tp[i] respectively.

Specifying Node Precomputed Functions Precompute statements for user-defined functions that compute a value at a mesh edge begin with the keyword 'PRECOMPUTE' and are followed by an 'at' (@) symbol, the keyword 'EDGE', an identifier, a parenthesized list of two comma-separated identifiers, the keyword 'ODD' or 'EVEN' an arrow, a function, and a semicolon. The first identifier is the name of a edge index. The second and third identifiers, which are parenthesized, specify the two nodes on the ends of the edge. These identifiers may be used by a specified user-defined function's arguments. The keywords 'ODD' or 'EVEN' specify that the user-defined function is either an odd or even function with respect to the order of the nodes. The syntax of these edge precompute statements is as follows.

```

PRECOMPUTE '@' EDGE identifier '(' identifier ',' identifier ')' ODD '->'
function ';'
```

```

PRECOMPUTE '@' EDGE identifier '(' identifier ',' identifier ')' EVEN '->'
function ';'
```

Examples of precompute statements for user-defined functions that compute a value at an edge are given below.

```

var V[NODES], n[NODES], p[NODES], C[NODES], tn[NODES], tp[NODES];

func E(V1,V2,<h>)
    return -grad(V1,V2,h);

func Jn(n1,n2,E,un1,un2,<h>)
    ...
    return un*(n*E+dndx);

precompute @ EDGE i (j,k) ODD -> E(V[j],V[k],elen);
precompute @ EDGE i (j,k) ODD ->
Jn(n[j],n[k],@E(i,j,k),@MUn(j),@MUn(k),elen);
```

The first precompute statement specifies that the user-defined function E is computed at edge i whose nodes are j and k with arguments $V1$, $V2$, and h equal to $V[j]$, $V[k]$, and $elen$ respectively. The keyword 'ELEN' represents the length of edge i . Note this function is odd, because $E(V[j],V[k],elen) = -E(V[k],V[j],elen)$. The last precompute statement specifies that the user-defined function Jn is computed at edge i whose nodes are j and k with arguments $n1$, $n2$, E , $un1$, $un2$, and h equal to $n[j]$, $n[k]$, $@E(i,j,k)$, $@MUn(j)$, $@MUn(k)$, and $elen$ respectively. The particular notation for $@E(i,j,k)$, $@MUn(j)$, and $@MUn(k)$ will be explained in the next section.

Using Precomputed Functions

Once a function has been specified for precomputation, the precomputed values of the function may be used in the body of equation statements and in other precompute statements. The syntax of precomputed functions is as follows.

```

'@' identifier '(' int expr ')'
```

```

'@' identifier '(' int expr ',' int expr ',' int expr ')'
```

The first syntax statement is for node precomputed functions. It begins with an 'at' (@) symbol and is followed by an identifier, and a parenthesized integer expression. The identifier is the name of the node precomputed function and the integer expression evaluates to the node index.

The second syntax statement is for edge precomputed functions. It begins with an 'at' (@) symbol and is followed by an identifier, and a parenthesized list of three comma-separated integer expressions. The identifier is the name of the edge precomputed function and the integer expressions evaluate to the edge and node indices.

For example, consider the last precompute statement below.

```
precompute @ EDGE i (j,k) ODD ->
Jn(n[j],n[k],@E(i,j,k),@MUn(j),@MUn(k),elen);
```

The third, fourth, and fifth arguments of the user-defined function Jn are @E(i,j,k), @MUn(j), and @MUn(k). @E(i,j,k) represents the precomputed value of the user-defined function E evaluated at edge i with nodes j and k. @MUn(j) and @MUn(k) represent the precomputed values of the user-defined functions MUn and MUp evaluated at nodes j and k respectively.

As an example of how precomputed functions may be used in an equation statement, consider the SGFramework code below.

```
equ n[i=SI] ->
+nsun(i,j,all,{@Jn(edge(i,j),i,node(i,j))}*ilen(i,j)) -
@R(i)*area(i) = 0.0;
```

@Jn(edge(i,j),i,node(i,j)) represents the precomputed value of the user-defined function Jn evaluated at edge edge(i,j), whose nodes are i and node(i,j).

This concludes the discussion of interfacing the equation and mesh specification files. We will now turn our attention to the executable programs and scripts which comprise the SGFramework.

1.4 SGFramework Executables

This section briefly describes the executable programs and scripts that form the SGFramework. (The SGFramework executables are invoked from a command line, so Windows95 and NT users must open an MS-DOS prompt to run the SGFramework executables, whereas in UNIX, a shell must be opened.) It should be apparent that it is all too easy to set up simulations which will fail, for various reasons. If a simulation does not converge, this may lead to overflows or to domain errors. These will often be reported as such, or they may cause the simulation to crash.

The SGFramework is schematically represented by Figure 1.4. The circles in the flowchart represent executable programs and scripts (batch files). This section is divided into several subsections. Each subsection describes an executable or script, lists its command-line options, and enumerates the warning and error messages that the executable may generate.

1.4.1 Build Script

The mesh parser and SGFramework translator produce C++ code. The code these programs generate must be compiled and linked with either the mesh refinement or SGFramework numerical algorithm module libraries. The SGFramework build script (sgbuild) facilitates the compiling and linking of the code generated by SGFramework programs.

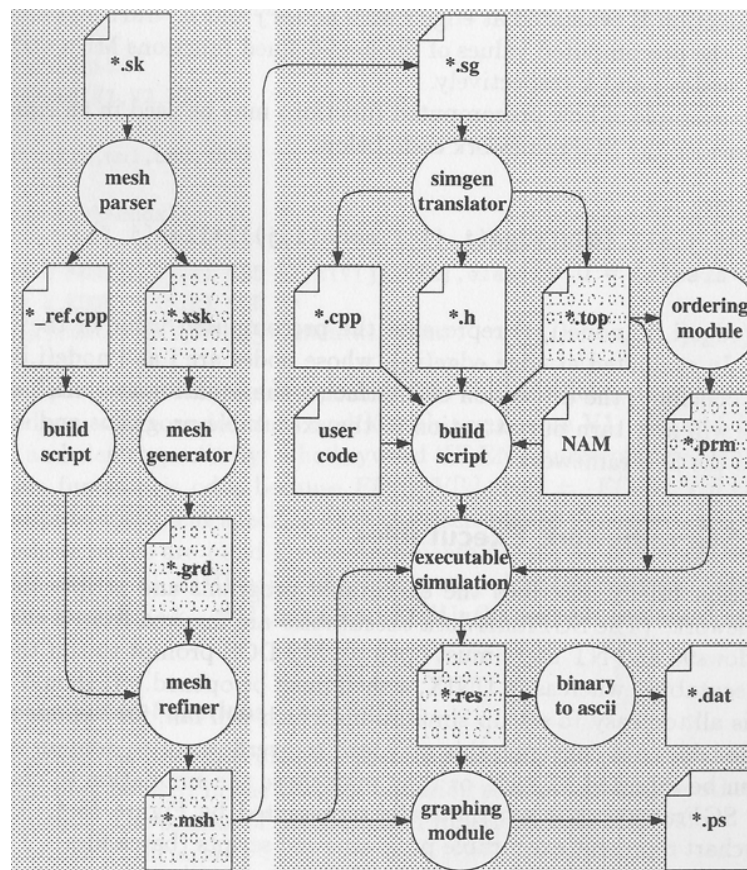


Figure 1.2. The overall flowchart for the SGFramework.

Command-Line Options

The SGFramework build script accepts two command-line arguments. The first argument is either 'ref' or 'sim' and the second argument is the name of the C++ source code to compile and link minus the extension. For instance, to compile the source code file `pin_ref.cpp` into a mesh refinement program, we would invoke the build script as follows.

```
sgbuild ref pin_ref
```

If the SGFramework build script is invoked with no or inappropriate command-line arguments, the following help information is displayed.

```
SGFramework Build Script      Version 1.0  Copyright (c) 1997 Kevin M. Kramer
* USAGE: sgbuild ref filename
*         sgbuild sim filename
*
* DESCRIPTION:
* sgbuild ref filename compiles and links the mesh refinement program
* sgbuild sim filename compiles and links the simulation program
*
* REMARKS:
* filename is the mesh refinement or simulation C++ source file name
* without the '.cpp' extension.
```

1.4.2 Mesh Parser

In any simulation which uses a complex mesh, we employ a mesh specification file to specify both the mesh and the way in which it is to be refined. The mesh specification file, which has an .sk extension, is parsed by the mesh program. This program generates a binary file with an .xsk extension and a C++ source file, which when linked with the mesh refinement library via the SGFramework build script, creates a mesh refinement program.

Command-Line Options

If the mesh parser (mesh) is invoked with no arguments or the -h command-line argument, the following help information is displayed.

```
SGFramework Skeleton Parser  Version 1.0  Copyright (c) 1995 Kevin M. Kramer
syntax: mesh [options] file
-h          display help screen on the standard output device
-p##       floating-point precision where ## = 1 to 16 (default = 6)
```

Warning and Error Messages

The mesh parser may generate the following warning and error messages. The format of the messages will be '[X] (row, col) message' where X is either a W for

warning, E for error, or F for fatal error; and row and col specify the row and column index of the place where the error occurred. Sometimes the actual error appears in the statement which precedes the one that is specified by the row and column indices. Because an error may cause several errors to follow, it is recommended that one fix the errors in the order they appear. In the explanation of the warning and error messages, all messages are referred to as errors even though they may only be warnings.

string cannot span multiple lines – Strings must begin and end with a double quote, both characters being on the same line.

unexpected end of file – The equation specification file ended unexpectedly. A common cause of this error is failure to specify the keyword ‘END’ at the end of the main procedure.

? expected – This error is caused by syntax errors. The question mark will be a list of valid characters or keywords that may appear at the specified position in the input file.

redefinition of symbol ? – The specified symbol has already been defined. For instance, one cannot define a constant named COUNT, and then declare a variable named COUNT.

symbol ? not defined – The specified symbol has not been defined. This error usually occurs when the name of a constant, function, variable, array or procedure has been misspelled.

? is not a ? symbol – The specified symbol is not of the required type. For instance, if an array is indexed by an expression enclosed in parentheses, rather than brackets, this error will occur, since the translator expects functions to be followed by a parenthesized list of expressions.

? symbol ? not declared – Because of a prior error, the specified symbol of the specified type has not been declared. This error will usually cause several additional errors to follow.

? not declared – This error is generated when the specified item (usually a parameter) has not been declared due to an error. This error usually occurs when a set statement attempts to initialize an unknown parameter.

coordinates not declared – This error is generated when the coordinates have not been generated due to an error.

? does not evaluate to a constant – The specified expression does not evaluate to a constant.

? does not evaluate to positive constant – The specified expression does not evaluate to a positive constant.

? variable does not evaluate to positive integer constant – The specified expression does not evaluate to a positive integer constant.

function ? has the wrong number of arguments – The specified function has the wrong number of arguments.

division by zero in ? – This error is caused when an expression or quantity is divided by another expression or quantity that evaluates to zero.

domain error in ? function – This error is caused when a function is called with arguments whose values are outside the domain of the function. For instance, taking the square root of a negative number will cause a domain error.

overflow in ? function – This error is generated when a precision overflow is generated.

underflow in ? function – This error is generated when a precision underflow is generated.

integer overflow – This error is caused when an integer expression evaluates to a value that is too large to represent as an integer.

cannot open file ? – The specified file cannot be opened.

cannot subdivide region ? into rectangular elements – The specified region cannot be divided into rectangular elements. Verify that the opposite edges have equal node spacing and that the edges are parallel to the coordinate axes.

1.4.3 Mesh Generator

The initial coarse grid is generated and plotted (unless specified) from a parsed mesh specification file. If a plot is generated, it is necessary to close the window before the next command can be accepted. Parsed mesh specification files have an .xsk extension.

Command-Line Options

If the mesh generator (sggrid) is invoked with no arguments or the -h command-line argument, the following help information is displayed.

```
SGFramework Grid Generator   Version 1.0   Copyright (c) 1995 Kevin M. Kramer
syntax:  gridgen [options] file                               * = default
-h          display this help screen
-Oxxx      create ASCII output file named xxx
-ee        * display errors on the standard error device
-eo        display errors on the standard output device
-t+        * exhaustive search for best triangulation
-t-        nonexhaustive search wherever appropriate
-i+        increase interior spacing
-i-        * do not increase interior spacing
-c+        * use centroid triangulation
-c-        do not use centroid triangulation
-g###      grade (1.5 to 2.5, default = 1.5)
-s###      maximum mesh spacing
-m###      minimum permissible quality (default 0.35)
-qe###     number of edge enhancement iterations (default 2)
-qn###     number of node enhancement iterations (default 2)
-qb###     number of edge followed by node enhancement iterations
(default 2)
-vr        * visualize mesh rating
-vn        no graphics
```

```

-ps      generate a postscript file named grid.ps
-r1      graphics screen resolution 480 x 640
-r2      * graphics screen resolution 800 x 600
-r3      graphics screen resolution 1024 x 768
-aspect  keep x and y aspect ratio when plotting

```

Warning and Error Messages

The mesh generation may generate the following warning and error messages.

out of memory – If this error message occurs, try closing some applications and try again.

file not found – The parser cannot locate the input file, verify the name and path of the input file and try again.

cannot create or open file – This error is caused when either the user does not have permission to open or create a file on the specified directory or the disk is full.

file write error – This error is caused when either the user does not have permission to write on the specified directory or the disk is full.

directory name too long – This error is caused when the directory name is too long. To remedy this error use a shorter directory name and try again.

all triangulation heuristics failed – The mesh generator is not able to triangulate the mesh. Verify that each region is specified by a simple, closed counterclockwise curve, the regions do not overlap, and that the segment spacing changes gradually (several short segments are not next to a long segment).

1.4.4 Mesh Refiner

In order to refine a mesh, the user must do two things. First the mesh refinement program must be compiled and linked. Second, the mesh refinement program must be executed. The mesh refinement program may be compiled and linked using the SGFramework build script. The mesh refinement source file is generated by the mesh parser. The name of the mesh refinement source file is 'xxxx_ref.cpp' where 'xxxx' is the first four characters of the mesh specification file from which the source file was generated. The mesh refinement program will generate a binary mesh file whose name is the name of the mesh specification file with a .msh extension. It may also plot the refined mesh (and the window containing the plot must be closed before the next command can be entered.)

Command-Line Options

If a mesh refinement program is invoked with no arguments or the -h command-line argument, the following help information is displayed.

```

SGFramework Mesh Refiner      Version 1.0  Copyright (c) 1995 Kevin M. Kramer
syntax: refine [options]                                * = default
-h          display this help screen
-Oxxx       create ASCII output file named xxx

```

```

-cyl          cylindrical geometry
-dmin###      minimum number of divisions
-dmax###      maximum number of divisions
-dadj###      divide triangles of level < ### if adj. triangle will be
split
-rm?=lin      use linear measurement for refinement variable ?
-rm?=log      use logarithmic measurement for refinement variable ?
-rm?=slog     use signed logarithmic measurement for refinement variable ?
-rd?=###      maximum delta for refinement variable ?
-qe###        number of edge enhancement iterations (default 2)
-qn###        number of node enhancement iterations (default 2)
-qb###        number of edge followed by node enhancement iterations
(default
2)
-m###         minimum rectangular splitting angle in degrees (default 0)
-g###         maximum grade between rectangular elements (default
infinite)
-s###         maximum ratio between length and width (default infinite)
-aspect       keep x and y aspect ratio
-vr           * visualize mesh rating
-vn           no graphics
-yz           do not apply refinement statements to elements above y = 0
-ps           generate a postscript file named mesh.ps
-bw           generate grayscale plot
-r1           graphics screen resolution 480 x 640
-r2           * graphics screen resolution 800 x 600
-r3           graphics screen resolution 1024 x 768

```

Warning and Error Messages

The Warning and Error Messages are:

cannot open file – This error is generated when the specified file could not be opened.

corrupt grid file – This error is generated when the grid (initial mesh) file is corrupt. To remedy this error, regenerate the initial mesh.

cannot create file – This error is generated when the specified file cannot be created.

1.4.5 SGFramework Translator

In order to generate a SGFramework simulation, a user must first translate an equation specification file via the SGFramework translator (using the command `sgxlat`). Then the user must compile and link the translator's output with a numerical algorithm module via the SGFramework build script (`sgbuild`).

Command-Line Options

If the SGFramework translator (sgxlat) is invoked with no arguments or the -h command-line argument, the following help information is displayed.

```
SGFramework Translator      Version 1.0  Copyright (c) 1995 Kevin M. Kramer
syntax:  sgxlat [options] file
  -h          display help screen on the standard output device
  -p##        floating-point precision where ## = 1 to 16 (default = 6)
  -nc         no coupling between equations
```

Warning and Error Messages

The SGFramework translator may generate the following warning and error messages. The format of the messages will be '[X] (row, col) message' where X is either a W for warning, E for error, or F for fatal error; and row and col specify the row and column index of the place where the error occurred. Sometimes the actual error appears in the statement which precedes the one that is specified by the row and column indices. Because an error may cause several errors to follow, it is recommended that one fix the errors in the order they appear. In the explanation of the warning and error messages, all messages are referred to as errors even though they may only be warnings.

string cannot span multiple lines – Strings must begin and end with a double quote, both characters being on the same line.

unexpected end of file – The equation specification file ended unexpectedly. A common cause of this error is failure to specify the keyword 'END' at the end of the main procedure.

? expected – This error is caused by syntax errors. The question mark will be a list of valid characters or keywords that may appear at the specified position in the input file.

redefinition of symbol ? – The specified symbol has already been defined. For instance, one cannot define a constant named COUNT, and then declare a variable named COUNT.

symbol ? not defined – The specified symbol has not been defined. This error usually occurs when the name of a constant, function, variables, array or procedure has been misspelled.

? is not a ? symbol – The specified symbol is not of the required type. For instance, if an array is indexed by an expression enclosed in parentheses, rather than brackets, this error will occur, since the translator expects functions to be followed by a parenthesized list of expressions.

? symbol ? not declared – Because of a prior error, the specified symbol of the specified type has not been declared. This error will usually cause several additional errors to follow.

? statement not declared – Because of a prior error, the specified statement of the specified type has not been declared. This error may cause several additional errors to follow.

user-defined function ? not declared – Because of a prior error, the specified user-defined function was not declared. This error will usually cause several additional errors to follow.

? does not evaluate to a constant – The specified expression does not evaluate to a constant.

array ? has too many dimensions – The specified array has too many dimensions. This error is generated if an array is declared that has over three dimensions.

dimension ? of array ? does not evaluate to a integer constant – The specified dimension of the specified array does not evaluate to an integer constant.

dimension ? of array ? is not an integer expression – The specified dimension of the specified array is not an integer expression.

dimension ? of array ? must be greater than zero – The specified dimension of the specified array must evaluate to an integer that is greater than zero. In other words, one cannot declare an array with zero or a negative number of elements.

dimension ? of array ? is out of bounds – The specified dimension of the specified array is out of bounds. In other words, the expression evaluates to an integer which is less than zero or greater than the number of elements in the specified dimension minus one.

array ? has the wrong number of dimensions – The specified array has the wrong number of dimensions. For instance, if a two-dimensional array is declared, the use of this array must be accompanied by a bracketed list of two-index expressions.

function ? has the wrong number of arguments – The specified function has the wrong number of arguments.

? value of index ? does not evaluate to an integer constant – the start, end, or step value of the specified index does not evaluate to an integer constant.

? value of index ? is illegal – the start, end, or step value of the specified index is illegal.

equation corresponding to ? ? is useless – The equation corresponding to the specified variable or array is not used because the variable or the unknown elements of the specified array are already associated with one or more equations.

equation corresponding to ? ? has no pivot – The equation corresponding to the specified variable or array has no pivot. This error results when the variable or array elements specified by the equation header do not appear in the equation's expression.

no main procedure statements specified – The equation specification does not have the mandatory main procedure.

no equation statements specified – No equation statements are declared in the equation specification file yet a solve statement is present.

division by zero in ? – This error is caused when an expression or quantity is divided by another expression or quantity that evaluates to zero.

domain error in ? function – This error is caused when a function is called with arguments whose values are outside the domain of the function. For instance, taking the square root of a negative number will cause a domain error.

overflow in ? function – This error is generated when a precision overflow is generated.

underflow in ? function – This error is generated when a precision underflow is generated.

integer overflow – This error is caused when an integer expression evaluates to a value that is too large to represent as an integer.

corrupt or missing mesh file – The specified mesh file is either corrupt or missing. To remedy this error, regenerate the mesh file.

near singularity encountered – This error is generated when the absolute value of a pivot is less than the specified ‘near singularity’ quantity.

unknown ? does not have a corresponding equation – The unknown variable or array element does not have a corresponding equation. Either add an equation or equations to the equation specification file or declare this variable or array element to be known.

underspecified system of equations – The system of equations is underspecified. A common cause of this error is when one equation is just a multiplicative constant times another equation.

overrelaxation parameter not in range [1.0, 2.0] – The overrelaxation parameter is not within its required range of 1.0 to 2.0.

symbol ? is not allowed in user-defined function body – The specified symbol is not allowed in the body of a user-defined function. A common cause of this error is using a variable or array element in the function’s body.

function ? may not call itself – This error is called when a function tries to recursively call itself. Recursion is not supported by the SGFramework translator.

a file must be open for ? prior to using this statement – This error is generated when the specified statement is used prior to opening an input or output file.

file already open, must issue a close statement – The SGFramework only supports one open file at a time. If the user tries to open a second file prior to closing the first file, this error will be generated.

cannot open file ? – The specified file cannot be opened.

the data file has not been closed – An open data file has not been closed prior to exiting the simulation.

unexpected end of data file – The end of a data file has been unexpectedly reached. A common cause of this error is searching for a label that does not exist in the data file.

syntax error in data file – A syntax error was encountered in the data file.

could not find label ? in data file – The specified label was not found in the data file.

main procedure already declared – An additional main procedure is present in the equation specification file.

? is not an integer expression – The specified expression is not an integer expression.

? function cannot contain another ? function – The first specified function cannot contain the second specified function. This error is usually caused by trying

to nest a summing function inside another summing function.

irregular mesh has not been declared – This error is generated when a mesh connectivity function or mesh geometry function is used but a mesh has not been imported.

the ? operator cannot be used in edge precompute statements – The specified operator cannot be used in edge precompute functions.

cannot determine if argument is odd or even w/ respect to its indices – This error is generated when the SGFramework translator cannot determine if an argument of a function is odd or even with respect to its indices.

1.4.6 Ordering Module

In order to increase the speed and reduce the memory requirements of SGFramework simulations, a user should order a simulation's unknown variables and array elements via the SGFramework minimum degree ordering program (order). The ordering program requires two command-line arguments: a SGFramework topology file and a SGFramework permutation file. The SGFramework topology file is generated by the SGFramework translator. The SGFramework permutation file will be generated by the ordering program.

Command-Line Options

If the minimum degree ordering program (order) is invoked with no arguments or the -h command-line argument, the following help information is displayed.

```
Minimum Degree Order    Version 1.0 Copyright (c) 1995 Kevin M. Kramer
usage:  order [options] topfile permfile
  -h    display this help screen
  -q    run quietly
  -Pxxx generate a SMMS permutation vector named xxx
```

Warning and Error Messages

The Warning and Error messages are:

cannot open file ? – This error is generated when the specified file cannot be opened.

cannot create file ? – This error is generated when the specified file cannot be created.

1.4.7 SGFramework Simulations

SGFramework simulations are built by compiling and linking the code generated by the SGFramework translator with an appropriate numerical algorithm module.

Command-Line Options

If the simulation is invoked with the -h command-line argument, the following help information is displayed.

SGFramework Simulation, SimGen Copyright (c) 1994 K. M. Kramer

```

-h          display this help screen
-vs0        run quietly, i.e. do not output convergence information, etc.
-vs1        output abbreviated convergence information
-vs2        output full convergence information
-vl1        output abbreviated convergence information to log file
-vl2        output full convergence information to log file
-z+         zero the dx vector prior to each iteration
-z-         do not zero the dx vector prior to each iteration
-Lxxx       generate log file xxx
-i##        maximum number of Newton iterations
-a##        minimum Newton accuracy
-d##        minimum error-reducing damping factor (min. df = 2^-##)
-e##        maximum percentage change per Newton iteration
-s##        near singularity value
-li##       maximum number of SOR, PCG, etc. iterations
-la##       maximum SOR, PCG, etc. accuracy
-lf##       maximum fill level for Gaussian elimination (## > 5 = infinity)
-lr##       over-relaxation parameter
-lsor       use successive over-relaxation algorithm
-lge        use Gaussian elimination algorithm
-p##        ASCII output precision for real numbers
-n##        use ## norm (## = inf for infinite norm)

```

Warning and Error Messages

The Warning and Error messages are:

- cannot open file ? – This error is generated if the specified file cannot be opened.
- cannot find label ? in data file ? – This error is generated if the specified label cannot be found in the specified file.
- syntax error in data file ? – This error is generated when a syntax error is encountered in the specified data file.
- unexpected end of file in data file ? – This error occurs if the end of the specified data file is unexpectedly encountered. A common cause of this error is searching for a label that either does not exist or is present only before the location of the file pointer.
- near singularity encountered – This error is generated when the absolute value of a pivot is less than the specified ‘near singularity’ quantity.

1.4.8 Extract Program

The binary-to-ASCII extraction program (extract) is used to extract data from a binary SGFramework result file. The data is written to ASCII files which are labeled by the name of the array whose value they contain. All variables are stored by the main program in a single file. The extension of these data files is determined by which snapshot of data is extracted. The command-line syntax of the extract program is as follows.


```
extract filename.res i
```

The number *i* is an integer that indicates that the *i*th set of data which was written to the .res file is to be extracted. Examples of shell scripts set up to perform the extraction are given in the text - see Section ??.

1.4.9 Group Program

The data file format program (group) is used to format unformatted data files generated by a SGFramework simulation. Often, several variables are repeatedly written to a data file. For instance, the mixed-mode simulation listed in Section ?? writes the variables t, dt, Vs, Vb, Vc, Vd, Vl, Ib, Ic, Il, and Id to the data file mm02.out at each time step via the file write statement. Since the file write statement outputs each variable on its own line, it is very difficult to analyze the data file mm02.out. A portion of the mm02.out file is shown below.

```
0
1e-09
12
0.5988724
24
4.572742e-07
```

The first number, 0, corresponds to t; the second number, 10^{-9} , corresponds to dt, and so on. It would be very hard to determine to what variable the hundredth or thousandth number corresponded. However, by using the group program, one can organize the data file into a list of columns. For example, consider the following invocation of the group program.

```
group t dt Vs Vb Vc Vd Vl Ib Ic Il Id <mm02.out >mm02.tab
```

The above example will group the data in the file mm02.out into eleven columns named t, dt, Vs, Vb, Vc, Vd, Vl, Ib, Ic, Il, and Id. (The command-line arguments are taken as the names of the columns, hence the number of command-line arguments is the number of columns into which the data is organized.) The SGFramework data file is redirected to the standard input device via the < symbol. The output, which is the organized (grouped) data, is redirected to the file mm02.tab. A portion of mm02.tab is shown below. Note that only the first five columns are shown.

t	dt	Vs	Vb	Vc
0.00000e+00	1.00000e-09	1.20000e+01	5.98872e-01	2.40000e+01

1.00000e-09	1.50000e-09	1.19760e+01	5.98818e-01	2.57290e+01
2.50000e-09	2.25000e-09	1.19400e+01	5.98736e-01	2.77730e+01
4.75000e-09	3.37500e-09	1.18860e+01	5.98613e-01	3.05299e+01
8.12500e-09	5.06250e-09	1.18050e+01	5.98427e-01	3.40187e+01
1.31875e-08	7.59375e-09	1.16835e+01	5.98146e-01	3.73361e+01
2.07812e-08	1.00000e-08	1.15013e+01	5.97718e-01	3.92992e+01

1.4.10 Graphical Output

The SGFramework provides mechanisms by which to view and print SGFramework meshes. Both the mesh generation program and the mesh refinement programs are capable of generating mesh plots. Furthermore, if these programs are invoked with the `-ps` command-line argument, these programs can generate postscript files of the mesh. These postscript files may be viewed and printed with postscript viewing applications.

Because it is not possible, in general, to solve semiconductor problems on regular meshes, and because most plotting software is only useful for regular meshes, SGFramework has its own plotting capabilities. The SGFramework surface plotter (triplot) is capable of plotting the results of a SGFramework simulation that used a mesh specification file. triplot requires one command-line argument, the name of a SGFramework result file without the extension. In addition, the user may specify an array to plot. The array must be one-dimensional and have as many elements as the mesh has nodes. If the name of an array is not specified on the command line, the user will be prompted to enter an array. Finally, the user may specify optional command-line arguments. For instance, `-log` plots the log of the values of the specified array (to be precise, $\text{sign}(x) \cdot \log(\text{abs}(x))$), `-az` specifies the azimuthal viewing angle of the plot, etc.

The SGFramework also contains another graphing program (sgplot). sgplot is similar to triplot in that it generates surface plots on irregular meshes. However, sgplot can also produce contours and it provides axes with tick marks. Invoke sgplot with the `-h` command line argument for instructions.

To make a postscript file of the results, the command line option `-ps` should be specified. The postscript file will have the same name as the simulation, with a `‘.ps’` on the end. In this book, for example, this command was used to plot the voltage from the simulation file `pn05.sg`. The command thus read

```
triplot pn05 V -ps
```

In this case, the SGFramework surface plotting program will generate a postscript file named `pn05.ps`. If there is more than one set of data (say from multiple time steps) the `-i###` command-line option should be specified to choose which data set to use. For instance, `-i1` uses the first data set, `-i2` the second, and so on. If the `-i###` command line option is not specified, the default is to use the first data set. This plotting routine makes use of a SGFramework mesh file, so it will not work in cases where we did not generate such a file.

Command-Line Options

If the SGFramework surface plotter (triplot) is invoked with no arguments or the -h command-line argument, the following help information is displayed.

```
SGFramework Surface Plotter  Version 1.0  Copyright (c) 1995 Kevin M. Kramer
syntax:  refine filename [array] [options]          * =
default
-h          display this help screen
-i###       use the ###'th data set (default 1)
-az###      azimuthal angle (default 30)
-el###      elevation angle (default 60)
-ps         generate a postscript file
-wf         wire frame
-bw         black and white (grayscale) plot
-nm         do not plot the mesh
-lin        * use a linear scale
-log        use a logarithmic scale
-r1         graphics screen resolution 480 x 640
-r2         * graphics screen resolution 800 x 600
-r3         graphics screen resolution 1024 x 768
```

Warning and Error Messages

cannot open file ? – This error is generated when the specified file cannot be opened.