Using and Programming the **SuperCode**

Scott W. Haney

July 21, 1995

Contents

Chapter 1

Introduction

1.1 Installing the Code

If you have an account on the M-Division server *moonshine* it is quite easy to install the SUPERCODE sources in your directory space. Only a few additional steps are required to get things working on another Sun workstation.

There are two ways to install the sources on *moonshine*: (1) Install *all* of the sources, including the sources for the *SCL* and *Shell* libraries, and for the various utilities; or (2) Install only the *SC* directory, which contains the physics and engineering modules. Most users working on *moonshine* should use the latter setup since it cuts down on the amount of time required to build the SUPERCODE from scratch, and it wastes less disk space.

The following subsections give step-by-step instructions for building the SUPERCODE. In the examples, the `typewriter` typeface will be used to indicate text appearing on the computer screen. Some of the examples will show the complete line, including the prompt:

```
moonshine[1]: cd $HOME
```

The prompt, "`moonshine[1]:`," symbolizes the prompt which appears at your terminal. Yours will most likely be different. Note that these example prompts contain a number, unique to each line in the example, which may be referred to in the text.

If there are any problems, please call Scott Haney at (510) 423–6308 or Jim Crotinger at (510) 422–0259.

1.1.1 Installing on Moonshine—the Entire Code

To install the entire code on *moonshine*, follow these easy steps:

1. Go to the directory where you want to install the SuperCode. For example:

   ```
   moonshine[1]: cd $HOME
   ```

2. Check out the SuperCode source tree:

   ```
   moonshine[2]: cvs -d ~haney/CVS co SuperCode
   ```

   You should end up with a directory named *SuperCode* that is full of files.

3. Add the following lines to your `$HOME/.cshrc` file:

   ```
   setenv CPU SUN4
   setenv CVSROOT /home/moonshine/haney/CVS
   ```

   You should also add `./SUN4` to your path. Look for the "`set path`" line in the .cshrc file, and add `./SUN4` somewhere near the beginning. For example:

   ```
   set path=(. ./SUN4 ~/bin <rest-of-path>)
   ```

   An example `.cshrc` file is contained in `Docs/scode.cshrc`. After adding these lines, type

   ```
   moonshine[3]: source $HOME/.cshrc
   ```

4. Build the SuperCode:

   ```
   moonshine[4]: cd SuperCode
   moonshine[5]: MakeImake
   moonshine[6]: MakeTopMakefile
   moonshine[7]: make World
   ```

The *make* will take roughly 20 minutes, depending on the load on the machine.

5. You are now the proud owner of a new SuperCode! To run it, type

```
moonshine[8]: cd SC
moonshine[9]: SuperCode
```

1.1.2 Installing on Moonshine—SC Only

Install only the `SC` portion of the code on *moonshine* as follows:

1. Go to the directory where you want to install the *SC* directory. For example:

```
moonshine[1]: cd $HOME
```

2. Check out the *SC* directory:

```
moonshine[2]: cvs -d ~haney/CVS co SC
```

You should end up with a directory named *SC* that is full of files.

3. Add the following lines to your `$HOME/.cshrc` file:

```
setenv CPU SUN4
setenv CVSROOT /home/moonshine/haney/CVS
```

You should also add `./SUN4` to your path. Look for the "`set path`" line in the .cshrc file, and add `./SUN4` somewhere near the beginning. For example:

```
set path=(. ./SUN4 ~/bin <rest-of-path>)
```

After adding these lines, type

```
moonshine[3]: source $HOME/.cshrc
```

4. Build the SUPERCODE:

```
moonshine[4]: cd SC
moonshine[5]: /usr/local/SuperCode/bin/MakeTopMakefile
moonshine[6]: make Makefiles
moonshine[7]: make
```

   The *make* will take roughly 20 minutes, depending on the load on the machine.

5. You are now the proud owner of a new SUPERCODE! To run it, type

```
moonshine[8]: SuperCode
```

6. If you want the *Docs* directory, you can also check it out:

```
moonshine[9]: cvs -d ~haney/CVS co Docs
```

1.1.3 Installing on another Sun SPARCstation

Install the code on another Sun SPARCstation as follows:

1. Log on to *moonshine*.

2. Go to a directory where you would like to check out a copy of the SUPERCODE. For example:

```
moonshine[1]: cd $HOME
```

3. Check out the SUPERCODE source tree:

```
moonshine[2]: cvs -d ~haney/CVS co SuperCode
```

   You should end up with a directory named *SuperCode* that is full of files.

4. Tar up the *SuperCode* for network transport:

```
moonshine[2]: tar cf sc.tar SuperCode
```

5. Log on to your SPARCstation and ftp *sc.tar*.

   ```
   mymachine[1]: ftp moonshine.llnl.gov
   ftp> bin
   ftp> get sc.tar
   ftp> ^D
   ```

6. Add the following line to your `$HOME/.cshrc` file:

   ```
   setenv CPU SUN4
   ```

   You should also add `./SUN4` to your path. Look for the "`set path`" line in the .cshrc file, and add `./SUN4` somewhere near the beginning. For example:

   ```
   set path=(. ./SUN4 ~/bin <rest-of-path>)
   ```

   An example `.cshrc` file is contained in `Docs/scode.cshrc`. After adding these lines, type

   ```
   mymachine[2]: source $HOME/.cshrc
   ```

7. Build the *imake* utility:

   ```
   mymachine[3]: cd SuperCode
   mymachine[4]: MakeImake
   ```

8. If you are still using Sun C++ 2.0, you must edit the imake configuration file for the Sun:

   ```
   mymachine[5]: emacs Utilities/Imake/config/sun.cf
   ```

   Look for the line defining `CPlusPlusMinorVersion` and change the definition from `1` to `0`. Also, the path required to find C++ may be different than that in *cshrc.scode*.

9. Build the SuperCode:

```
mymachine[6]: MakeTopMakefile
mymachine[7]: make World
```

The last command *make* will take roughly 20 minutes, depending on the load on the machine.

10. You are now the proud owner of a new SUPERCODE! To run it, type

```
mymachine[8]: cd SC
mymachine[9]: SuperCode
```

1.2 Directory Organization

Users who install the entire SUPERCODE distribution will be presented with over 200 source files and 25 sub-directories. They are laid out as follows:

```
SuperCode:
    Docs:
    RLStream:
        SUN4:
    SC:
        SUN4:
    SCL:
        SUN4:
    Shell:
        SUN4:
    Utilities:
        Imake:
            config:
                SuperCode:
            src:
                SUN4:
        MGen:
            SUN4:
        Mppl:
            SUN4:
        Readline:
            SUN4:
```

The *SuperCode/Docs* directory contains the LaTeX files used to generate this manual. The manual can be produced by typing "`latex SCManual`". The *SuperCode/SC* directory is of most interest to module contributors. This is where the source code for modules goes and where the SuperCode can be executed. There are also several example shell language programs here. The *SuperCode/SCL* directory contains a C++ class library used to build the SuperCode. The *SuperCode/RLStream* and *SuperCode/Utilities/Readline* directories contain files used to implement the SuperCode command line editing features. The *SuperCode/Shell* directory contains the C++ source code for the shell. The *SuperCode/Utilities/Imake* directory contains files used by the *imake* utility. The *SuperCode/Utilities/MGen* directory contains the source for the SuperCode module description file reader. Finally, the *SuperCode/Utilities/Mppl* directory contains the source for an enhanced version of MPPL used in module development. The various *SUN4* directories hold object files and the like. Hopefully, this brief discussion will help with navigation through the directory structure.

Chapter 2

The Shell

2.1 Executing the Code

Until the graphical front-end is developed, the SuperCode is run in a command-line mode. This version of the SuperCode can be most simply executed by typing

```
SuperCode
```

After a bit, the SuperCode shell responds with its prompt

In[1]:

In this manual, we will always use **bold face** to represent text the computer types and `typewriter face` to represent text the user types. The number in the prompt refers to line number of interactive input currently being entered. The SuperCode is an interactive code so a run consists of alternating user inputs and code results. For example,

```
In[1]:
1 + 1;
  2
In[2]:
2 * (3 + 4);
  14
In[3]:
quit
```

The `quit` directive is used to terminate the session.

It is possible to read statements into the SUPERCODE via files. For example, if the file *test.sc*[1] contained the lines

```
1 + 1;
2 * (3 + 4);
```

and the SUPERCODE was executed with the command

```
SuperCode test.sc
```

the SUPERCODE would respond as if the user had typed the lines in *test.sc* directly at the terminal:

```
  2
  14
In[1]:
```

It is also possible to read files into the SUPERCODE at any time during a session using the `include` directive

```
In[1]:
include "test.sc"
  2
  14
In[2]:
```

Notice that double quotation marks around the file name are required here. For compatibility with the C language convention, `#include` can be used in place of `include`.

Files can contain `include` directives. In this case, processing of the current file is temporarily suspended while the `include`'d file is read in. A file can also contain a `quit` directive. By placing a `quit` directive in a file, the SUPERCODE can be executed in a batch mode similar to conventional codes.

When the SUPERCODE starts up, it looks for a file 'SuperCode.sc'. If it can find it, the SUPERCODE reads the statements in that file before any user-specified files. This start-up file is useful for defining variables and routines are should exist for every run to the SUPERCODE.

In any interactive code, errors are bound to occur. The SUPERCODE shell is capable of detecting exceptions and returning to a state ready to accept new input. For example

[1]By convention, SUPERCODE shell statement files possess the suffix `.sc`.

In[1]:
```
1 + * 1;
```
Error 12: Misplaced token.
I wasn't expecting to see the operator '*' here.
File "Interactive"; Line 1 # caused the exception.
File "Expression.cc"; Line 1072 # generated the exception.
In[2]:

The first few lines describe the type of exception. Then, the file and line number of the shell language instruction that actually caused the exception is given[2]. Finally, the file and line number of the instruction in the SUPERCODE source that generated the exception is given. This last piece of information is of most use to SUPERCODE implementors.

Three types of exceptions can be generated. These are, in order of increasing severity:

- *Warnings*. Indicating a potential problem that is not serious enough to stop execution of the current set of shell statements.
- *Errors*. Indicating a problem that requires the execution of the current set of shell statements to cease.
- *Fatals*. Indicating a problem that requires the execution of the SUPERCODE to cease.

Fatal errors are generally indicative of a programming error. Therefore, please note the error message along with what you were typing when the fatal error occurred and contact Scott Haney at (510)–423–6308.

2.2 Command Line Editing and History

The SUPERCODE shell allows the user to use the cursor keys and control keys to edit the line which he is typing and to recall lines which have been previously entered. Here is a summary of the features

- Use the up and down arrows (or the EMACS control sequences, Control-P and Control-N) to scroll back and forth through lines previously entered.

[2] File "Interactive" refers to interactive input.

- Left and right arrows (or Control-B and Control-F) can be used to move within the text of a given command.
- Control-A moves to the beginning of a command, Control-E moves to the end.
- Control-R starts an EMACS-style search back through the previously entered commands. Control-S searches forward (which is only useful if you've already moved back in the buffer).
- Try out other EMACS editing keys such as Control-K, Control-Y, Control-D, Esc-D, etc. Most work.
- Hit the TAB key after typing the beginning of an include file name and the SuperCode will try to complete it for you. For example, typing "`include "cd`" followed by a TAB will cause the SuperCode to finish the string, resulting in "`include "cda.sc"`" appearing at the prompt.

The command line editing and history features were added to the SuperCode shell via the GNU Readline Library from the Free Software Foundation (FSF). Full documentation on these features are described in detail in the FSF's *GNU Readline Library Manual*. If you are interested in obtaining these, please contact Scott Haney or Jim Crotinger.

2.3 The Shell Language

Conventional codes typically read input files consisting of a list of data values. The SuperCode, on the other hand, reads input consisting of a list of executable statements. These statements are high-level constructs that allow

- Declaration of variables of various types.
- Arithmetic involving variables and constants.
- Tests and looping.
- Subroutines and functions.
- Access to compiled computational routines and modules.

The SuperCode shell language is based on the C++ programming language [1], which was developed by Bjarne Stroustrup in 1985. C++, in turn, is an extension of the popular C programming language [2], developed by Dennis Richie and Brian Kernigan in the early 1980s.

The balance of this section will briefly describe the main features of the SuperCode shell language. Despite the fact that the shell language differs significantly from FORTRAN, this section, along with a little bit of actual practice with the SuperCode should allow physicists and engineers to quickly pick up the syntax. If further information is required, please see one of the numerous books on C [3] or C++ [4, 5, 6]. In many cases, C or C++ examples from these books can executed directly in the shell.

2.3.1 Input Format

Unlike FORTRAN, which is sensitive to statement spacing, the shell language is completely free-format. Accordingly, statements can be spread over several lines and spaces, tabs, etc are generally unimportant. For example,

```
In[1]:
1 +
1
;
  2
In[4]:
```

is ugly but completely legal. This flexibility comes at a small cost though: a semi-colon is usually required to signal the end of a source line. In the above example, notice how the result of the calculation and the new prompt are not displayed until `1 + 1;` is completely entered. The shell will patiently wait until either a correct shell language statement or a syntax error is entered. If you have typed something and you think the shell should be doing something, it is probably waiting for the semi-colon you forgot to type.

2.3.2 Comments

To aid in documentation, the shell recognizes comments. For example the sample file *test.sc* discussed above can be modified to

```
/*
 * test.sc:  a very simple shell program
 */

1 + 1; // A very simple calculation.
2 * (3 + 4);
```

There are two types of comments allowed by the shell and both are present in the modified file. The first type delimits the text of the comment with the tokens `/*` and `*/`. Everything between these tokens is counted as a comment and ignored by the shell. The second type, signaled by the `//`, indicates that the text following this token to the end of the line is a comment. Comments can also be typed during interactive input

```
In[1]:
/*
 * test.sc:  a very simple shell program
 */

1 + 1; // A very simple calculation.
  2
In[6]:
2 * (3 + 4);
  14
In[7]:
```

Note that the shell will not return a new prompt until an executable statement is given.

2.3.3 Declarations

FORTRAN's implicit typing rules generally make it unnecessary to explicitly specify the types of variables. This has the advantage of reducing typing but has the disadvantage of being error-prone. For example, forgetting that a variable named `mu0` is, by default, an integer, causes big problems when that variable is meant to represent the permittivity of free space ($4\pi \times 10^{-7}$). To combat this sort of bug, the shell language requires that each and every variable be declared to have a particular type.

Basic Types

There are two fundamental types of variables: `Real` and `Integer`. The simplest types of declarations are written

```
Real a;
Integer i;
```

These two declarations create the real variable `a` and the integer variable `i`. These variables can be assigned values and manipulated in arithmetic calculations:

In[1]:
```
a = 2.3;
```
In[2]:
```
a * (2 + 1.2);
```
  7.36

In[3]:
```
i = a + 27;
```
In[4]:
```
i;
```
  29

Shell variable names must start with either a letter or an underscore. After that, any combinations of letters, numbers, and underscores can follow. The maximum variable name length is currently 256 characters. Unlike FORTRAN, the shell *is* sensitive to case so `a` is a different variable than `A`.

More elaborate declarations are possible. For instance, one can declare several variables at once by listing the variable names separated by commas

```
Real x, y, z;
Integer l, m;
```

Sometimes, it is also desirable to initialize variables directly in the declaration

```
Real a = 2.3 + (2 * 1.2);
Integer i = a + 27;
```

Finally, one also occasionally wishes to declare a variable that should not be modified. This can be done by prepending the `const` keyword to the beginning of the declaration

```
const Real pi = 3.14159;
```

Variables declared in this manner cannot appear on the left hand side of an equals sign. If you try to assign to a `const` variable, an error is generated. As a result, `const` variables *must* be initialized when they are declared (if you forget, the shell will generate an error).

The shell also supports a fairly unique type of variable called a *reference*. Reference variables are declared by placing an ampersand in front of the variable name. Therefore, the declaration

```
Real &aref = a, b;
```

declares a reference variable `aref` and an normal real variable `b`. A reference is an alias for another variable. In the example above, `aref` is a reference to the variable `a`. This means that all operations done with and to `aref` are equivalent to performing those operations with or to `a` (and visa versa):

```
In[1]:
Real a = 4.12, &aref = a;
In[2]:
aref;
  4.12
In[3]:
aref = 9.58;
In[4]:
a;
  9.58
In[4]:
a = 2.01;
In[5]:
aref;
  2.01
```

A non-`const` reference cannot be an alias for a constant. However, it is possible declare a `const` reference that is. It is also illegal to have a non-`const` reference variable alias a variable of a different type. Here are some examples:

```
Integer i;
Real a;
Integer &j = i; // Legal
Integer &k = 1; // Illegal
```

```
const Integer &k = 1; // Legal
Integer &l = a; // Illegal
const Integer &l = a; // Legal
```

You may be wondering what reference variables are actually good for. A later section will describe how they are extremely useful as function or subroutine parameters.

For convenience, the shell automatically declares a number of work variables including the `Integers` `_i`, `_j`, `_k`, `_l`, `_m`, `_n` and the `Reals` `_a`, `_b`, `_c`, `_d`, `_e`, `_f`.

Other Types

In addition to real numbers and integers, the shell currently supports a variety of other types including

- `Complex`. Complex numbers.
- `ComplexArray3`. Three-dimensional arrays of complex numbers.
- `ComplexMatrix`. Two-dimensional arrays of complex numbers.
- `ComplexVector`. One-dimensional arrays of complex numbers.
- `Fstream`. Stream input or output from files.
- `IntegerArray3`. Three-dimensional arrays of integers.
- `IntegerMatrix`. Two-dimensional arrays of integers.
- `IntegerVector`. One-dimensional arrays of integers.
- `Ostream`. Stream output to the terminal.
- `RealArray3`. Three-dimensional arrays of real numbers.
- `RealMatrix`. Two-dimensional arrays of real numbers.
- `RealVector`. One-dimensional arrays of real numbers.
- `String`. Character strings.

- `Subroutine`. Aliases for subroutines taking no arguments.
- `Void`. No type.

These types will be described in more detail later in this chapter.

Functions

In addition to variables, the shell allows the declaration of sophisticated functions and subroutines. Function declarations are similar to variable declarations except

- Only one function can be declared at once.
- The types of arguments the function expects to get must be specified.
- The statements making up the function must be supplied.

For example, the declaration

In[1]:
```
Real foo(Integer j, Integer k)
{
  ⋮
}
```

says that `foo` has the type "function expecting two `Integer` arguments and returning a `Real`". The two variables `j` and `k` are known as *formal arguments*. The unspecified stuff inside the curly brackets is the statements the function is supposed to execute when it is called. The return type can be any of the possibilities we just discussed (including a reference). Similarly, there can be any number of arguments of any type. Note that even if there are no arguments, a pair of parentheses must follow the function name.

To declare a subroutine have it return type `Void`:

In[1]:
```
Void bar(const Real &xyzzy)
{
  ⋮
}
```

This is the only place `Void` can appear.

A number of useful arithmetic functions have been pre-defined including `sqrt`, `sin`, `cos`, `tan`, `asin`, `acos`, and `atan`.

Subroutine Aliases

On occasion, it is very convenient to have a variable that serves as an alias of a function. Then, this variable can be passed to another function and used to call the original function. The shell currently has a limited capability to do this using the `Subroutine` type. The `Subroutine` type is an alias to functions of the form

```
Void vFun()
{
  ⋮
}
```

and a `Subroutine` variable can be instantiated with a declaration of the form

```
Subroutine aAa = vFun;
```

After this declaration, `aAa` can be used interchangedly with `vFun`. Unlike references, which have similar sorts of properties, it is possible to assign a new function to alias after creation

In[1]:
```
Void vFun() { }
```
In[2]:
```
Void vFun1() { }
```
In[2]:
```
Subroutine aa, bb = vFun1;
```
In[3]:
```
aa = bb; bb = vFun;
```
In[4]:

We will see all sorts of uses for `Subroutine` variables in later chapters.

2.3.4 Expressions

The shell understands a wide variety of arithmetic operators. These operators are summarized in Table 2.3.4. Note that all operators between a set of horizontal lines in the table have the same precedence. They also have the same grouping (or associativity) properties. As in FORTRAN, the order of evaluation can always be modified by grouping things in parentheses. The balance of this section will describe the operators in more detail.

Function Call

Function calls have the syntax

expression1(*expression-list*)

where *expression1* is an expression evaluating to "function returning `T`", `T` is some type (including `Void`), and *expression-list* is a comma separated list of expressions which constitute the actual arguments of the function. Note that *expression-list* may be empty but the parentheses are still required. Finally, there is no need for a `call` keyword even for `Void` functions (i.e., subroutines).

The simplest *expression1* is the name of a previously defined function

```
In[1]:
Integer i = 1;
In[2]:
Void foo(Integer j, Integer k)
{
  i = j + k;
}
In[6]:
foo(3, 5);
In[7]:
i;
  8
In[8]:
```

although more complicated situations can arise. For instance, suppose `a` is a function taking no arguments and returning a `Subroutine` that is an alias for a `Void` function `b`. Then, the surprising statement

```
a()();
```

Table 2.1: Shell Operator Summary

<table>
<tr><th>Operator(s)</th><th>Grouping</th></tr>
<tr><td>Function call ()
Postincrement ++
Postdecrement --</td><td>left to right</td></tr>
<tr><td>No operation +
Negation -
Logical negation !
One's complement ~
Preincrement ++
Predecrement --</td><td>right to left</td></tr>
<tr><td>Multiplication *
Division /
Remainder %</td><td>left to right</td></tr>
<tr><td>Addition +
Subtraction -</td><td>left to right</td></tr>
<tr><td>Left bit shift <<
Right bit shift >></td><td>left to right</td></tr>
<tr><td>Relational <, >, <=, >=</td><td>left to right</td></tr>
<tr><td>Equal to ==
Not equal to !=</td><td>left to right</td></tr>
<tr><td>Bitwise AND &</td><td>left to right</td></tr>
<tr><td>Bitwise Exclusive OR ^</td><td>left to right</td></tr>
<tr><td>Bitwise Inclusive OR |</td><td>left to right</td></tr>
<tr><td>Logical AND &&</td><td>left to right</td></tr>
<tr><td>Logical OR ||</td><td>left to right</td></tr>
<tr><td>Conditional ? :</td><td>right to left</td></tr>
<tr><td>Assignment =, *=, /=, %= +=, -=, <<=, >>= &=, ^=, |=</td><td>left to right</td></tr>
<tr><td>Comma ,</td><td>left to right</td></tr>
</table>

actually ends up calling `b`!

When a function is called, each formal argument of a function is initialized with the actual argument at the same location in *expression-list*. Inside the function, it is possible to change the values of any non-`const` formal argument but these changes will not affect the values of the actual arguments unless the arguments are references. This is different from FORTRAN where all changes to the formal arguments change the actual arguments. Here is an example:

```
In[1]:
Integer i = 1;
In[2]:
Void foo(Integer k) { k = 3; }
In[3]:
Void bar(Integer &k) { k = 3; }
In[4]:
foo(i);
In[5]:
i;
  1
In[6]:
bar(i);
In[7]:
i;
  3
In[8]:
```

This, incidentally, is the main use for references: they allow functions to modify the actual arguments passed to them.

As we mentioned in the last section, it is possible to declare two functions with the same name but taking different arguments. How does the shell decide which function to call? This decision is made by comparing the actual arguments with the formal arguments of all of the functions with the same name. The actual procedure is fairly complicated but it can be summarized as follows:

1. Eliminate all possibilities that expect the wrong number of arguments.

2. For each argument, find the set of functions whose formal argument best matches the actual argument. An exact match (e.g. a `Real` with a `Real`) is better that a conversion (e.g. a `Real` with an `Integer`). If a

function has formal arguments that cannot be initialized by the actual arguments (e.g., a `RealMatrix` formal argument cannot be initialized by an `Integer`) that function is eliminated from the competition.

3. Form the intersection of the sets of functions from each of the arguments that yield an equally good match. If the resulting set consists of one and only one function, this is the one to call. If the resulting set is empty, this is an error. If the resulting set has more than one function in it, the call is ambiguous and an error is generated.

This isn't as hard as it sounds. Consider the following

```
In[1]:
Integer i;
In[2]:
Void f(Integer l, Integer m) { i = 1; }
In[3]:
Void f(Real l, Real m) { i = 2; }
In[4]:
Void f(Integer l, Real m) { i = 3; }
In[5]:
f(1, 1);
In[6]:
i;
  1
In[7]:
f(1.0, 1.0);
In[8]:
i;
  2
In[9]:
f(1, 1.0);
In[10]:
i;
  3
In[11]:
f(1.0, 1);
  ### Error 40: Ambiguous call to overloaded function.
      Multiple versions of 'f' can accept the values passed.
      File "Interactive"; Line 12 # caused the exception.
```

File "TypeCheck.cc"; Line 957 # generated the exception.
In[12]:

The last call is ambiguous because `f(Real, Real)` has an exact match on the first argument and it requires an `Integer` $\longrightarrow$ `Real` conversion on the second while `f(Integer, Integer)` has an exact match on the second argument and it requires a `Real` $\longrightarrow$ `Integer` conversion on the first.

Postincrement and Postdecrement

These operators return the value of the operand and then add (`++`) or subtract (`--`) 1 from it. The operand must be what is called an *lvalue*. An lvalue is basically something that can appear on the left hand side of an equals sign. For example, a non-`const` variable is an lvalue while the number `3` is not. The result of a postincrement or decrement is not an lvalue.

Unary Plus

A unary plus in front of an operand just returns that operand. The result is not an lvalue.

Unary Minus

A unary minus in front of an operand returns the negation of that operand. The result is not an lvalue.

Logical Negation

The result of the unary logical negation operator (!) is of type `Integer`. If the operand is 0, the operator returns 1, if the operand is non-zero, the operator returns 0. The result is not an lvalue.

One's Complement

The operand of the one's complement operator ($\sim$) must be an `Integer`. The result is an `Integer` with all of the 0 bits in the binary representation of the operand switched to 1 and visa versa. The result is not an lvalue.

Preincrement and Predecrement

These operators add (`++`) or subtract (`--`) 1 from the operand and return than new value. The operand must be an lvalue and the result is not an lvalue.

Multiplication, Division, and Remainder

These are all binary operators that return non-lvalues. Multiplication (`*`) and division (`/`) work in the way you'd expect. The remainder operator (`%`) only works with operands of type `Integer`. The result is what is left over after dividing the first operand by the second. Therefore, `(a/b) * b + a % b` equals `a`.

Addition and Subtraction

These are binary operators that return non-lvalues. Addition (`+`) and subtraction (`-`) work as they always have.

Left and Right Bit Shifts

These are binary operators that expect `Integer` operands and return non-lvalues. These operators shift the bits of the left operand the number of places specified by the right operand to the left (`<<`) or right (`>>`). Both shifts fill in with zeros as the number is shifted. For example `1 << 3` equals 8 and `16 >> 2` equals 4.

Relational Operators

These are binary operators that return non-lvalues of type `Integer`. The possible comparisons are less-than (`<`), greater-than (`>`), less-than-or-equal (`<=`), or greater-than-or-equal (`>=`) They compare the two operands and return 1 if the condition evaluates to true and 0 if the condition evaluates to false. You can count on these values so relational comparisons can be used in arithmetic operations.

Equality Operators

These are binary operators that return non-lvalues of type `Integer`. The possible comparisons are equal-to (`==`) or not-equal-to (`!=`). They compare the two operands and return 1 if the condition evaluates to true and 0 if the condition evaluates to false. You can count on these values so equality comparisons can be used in arithmetic operations.

Bitwise Operators

These are binary operators that expect `Integer` operands and return non-lvalues. These operators perform the bitwise OR (|), bitwise exclusive-OR (`^`), or bitwise AND (`&`) boolean functions on the operands.

Logical AND

This binary operator (`&&`) returns a non-lvalue of type `Integer`. The result is 1 if both operands are non-zero and 0 if one of the operands is 0. If the first operand is zero, the second is *not* evaluated.

Logical OR

This binary operator (||) returns a non-lvalue of type `Integer`. The result is 1 if either operand is non-zero and 0 if both of the operands are 0. If the first operand is non-zero, the second is *not* evaluated.

Conditional Operator

This is the world's first ternary operator. It has the syntax

expression1 ? *expression2* : *expression3*

This operator works by first evaluating *expression1*. If this is non-zero the result is whatever *expression2* evaluates to. Otherwise, it is is whatever *expression3* evaluates to. For example, consider the expression

```
a = (b > 3) ? 12.5 : 1.25;
```

If `b == 4`, `a` is assigned 12.5, if `b == 1`, `a` is assigned 1.25.

Assignment Operators

The shell supports a bunch of assignment operators that all require an lvalue as the left operand and return an lvalue with the type of the left operand. The possible assignments are simple-assignment (`=`), multiply-and-assign (`*=`), divide-and-assign (`/=`), modulo-and-assign (`%=`), add-and-assign (`+=`), subtract-and-assign (`-=`), shift left-and-assign (`<<=`), shift right-and-assign (`>>=`), AND-and-assign (`&=`), inclusive OR-and-assign (`|=`), or exclusive OR-and-assign (`^=`). In simple assignments, the value of the right operand (after possible conversions) replaces that of the left operand. The behavior of the other assignments of the form

expression1 op`=` *expression2*;

is equivalent to

expression1 `=` *expression1 op expression2*;

except that *expression1* is only evaluated once.

Comma Operator

This binary operator (`,`) first evaluates the first operand and discards the result. Then, it evaluates the second operand and that is the result. The result is an lvalue if the second operand is an lvalue The effect of the first operand is felt via side effects. For instance, the expression

```
a = (b = 11, b + 7);
```

ends up setting `a` to 18 and `b` to 11. Be sure to apply parentheses liberally with the comma operator since it is the lowest precedence operator around. Also, take care when using it in function calls since the shell assumes the commas there separate the arguments.

2.3.5 Statements

The SUPERCODE shell interprets statements until the `quit` directive is issued. The statements that the shell understands are summarized in Table 2.3.5. The balance of this section will describe these statements in more detail.

Table 2.2: Shell Statement Summary

Category	Syntax
Declaration	*Discussed in previous section*
Null	;
Expression	*expression* ;
Selection	`if` (*expression*) *statement1* `if` (*expression*) *statement1* `else` *statement2*
Looping	`while` (*expression*) *statement* `for` (*expression1* ; *expression2* ; *expression3*) *statement*
Compound	{ *statements* }
Jump	`break` ; `continue` ; `return` ; `return` *expression* ;

Null Statement

A null statement consists of a single semi-colon. It is the simplest statement known by the shell. Null statements don't do anything but they are useful occasionally.

Expression Statement

Most of the statements you type into the shell will be expression statements. Expression statements consist of an expression (usually an assignment or a function call) followed by a semi-colon. All side effects of an expression statement are completed before the next statement is executed. We have already given numerous examples of this sort of statement so none will be supplied here.

Selection Statements

Selection statements choose a particular flow of control. Based on Table 2.3.5, we see that the simplest selection statement can look something like

```
if (a > 3 && a < 6)
  b = 47;
```

If the condition inside the parentheses is non-zero, the following statement (`b = 47;`) is executed, otherwise control is transferred to the statement following that.

It is also possible to add a statement to execute if the condition is zero. This is accomplished with the `else keyword`:

```
if (a > 3 && a < 6)
  b = 47;
else
  b = -12;
```

In this case, if `a == 4`, the variable `b` is assigned 47 whereas if `a == 1`, `b` is assigned -12.

The introduction of the `else` keyword creates an ambiguity when reading nested `if` statements. Consider

```
if (x <= 7 || x > 22)
  if (x == 33)
    y = -x * z;
else
  y = 3.5 * z + 4;
```

Despite the suggestive indentation, the `else` clause does not belong with the initial `if` statement; instead, it goes with the second. The rule is that an `else` always goes with the last `else`-less `if`. We will soon see how to achieve the flow of control indicated by the indentation.

There is a slight subtlety to remember when typing in `if` statements interactively. After shell sees the semi-colon after the expression `b = 47` it does not know whether or not an `else` clause will follow or not. Therefore, the shell will continue to wait for input in order to decide. If you don't have an `else` following the `if` statement, simply type a semi-colon (i.e., a null statement) and the shell will evaluate the `if`:

In[1]:
```
Integer i = 3, j;
```
In[2]:
```
if (i == 3)
```

```
  j = 4;;
```
In[4]:
```
j;
```
  4

The second semi-colon after the expression `j = 4` is the null statement.

Looping Statements

As Table 2.3.5 suggests, the shell understands two looping constructs: `while` statements and `for` statements.

An example of a `while` statement is

```
Integer i = 0, j = 0;
while (i < 5)
  j += i++;
```

In a `while` loop, the statement is executed repeatedly until the expression inside the parentheses after the `while` keyword evaluates to zero. This test takes place before the execution of the statement. Using these rules, it is fairly easy to determine that `j == 10` after the loop is executed.

The `for` loop is a more compact way of implementing the same algorithm as a `while` loop. An `for` loop equivalent to the `while` loop above is

```
Integer i, j;
for (i = 0, j = 0; i < 5; i++)
  j += i;
```

As you can see, there are three expressions inside the parentheses after the `for` keyword. The first expression specifies initialization for the loop (note the use of the ubiquitous comma operator). The second expression specifies a test made before each iteration. If this expression evaluates to zero, the loop is exited. The third expression is evaluated after each iteration. It usually serves to increment a counter.

Compound Statements

It would be very inconvenient if loops could consist of only one statement. Happily, the shell supports a construct called a *compound statement* that solves this problem. Simply put, a compound statement is a series of statements surrounded by curly brackets. Compound statements are very powerful because they can be used anywhere a statement can be used. Therefore, the `while` loop example above can be written in a slightly more readable form

```
Integer i = 0, j = 0;
while (i < 5)
  {
    j += i;
    i++;
  }
```

The bodies of functions are actually compound statements. Therefore, we can define a function having more than one line. For instance

```
Void swap(Real &x, Real &y)
{
  Real z;
  z = x;
  x = y;
  y = z;
}
```

is a subroutine that exchanges the value of two `Real` numbers.

Notice the presence of a declaration in the compound statement making up the body of `swap`. One can place any number of variable declarations anywhere inside a compound statement. However, function declarations cannot be placed inside a compound statement.

In addition to providing a way to group statements, a compound statement also introduces what is called a *scope*. Scope is used to determine the meaning of a name as it is used at a particular place in a shell program. Up to now, this has been obvious

In[1]:
```
Integer i;
```
In[2]:
```
Integer j = 2;
```
In[3]:
```
i = 3 + j;
```
In[4]:

Clearly, the expression in **In**[**3**] refers to the `i` declared in **In**[**1**] and the `j` in **In**[**2**]. However, suppose we have the situation

```
In[1]:
Integer i = 0;
In[2]:
Integer j = 2;
In[3]:
{
  Integer i;
  i = 3 + j;
}
In[7]:
```

What is the value of `i` after this set of statements? It turns out that the answer is `i == 0`.

The scoping rules used in the SUPERCODE shell are relatively simple

1. The scope of a name extends from the point of declaration to the end of the compound statement the declaration resides in. This is also the lifetime of any physical storage required for the variable.

2. A declaration of a name in a compound statement hides declarations of the same name from enclosing scopes.

3. After exit from the compound statement, the name resumes its previous meaning.

To understand these rules, consider the following code:

```
Integer jj = 6;
Void g(Integer kk)
{
  Integer ii = jj;
  Integer jj = 13;
  ii = jj;
}
```

The variable `ii` inside `g` is initialized using the global version of `jj`. Hence, its value is initially 6. When the local version of `jj` is declared, the global `jj` is hidden so when `ii` is set on the next line, it is set to 13. The storage for the local variables `ii` and `jj` exists until the closing curly bracket of `g`. The same goes for the formal parameter `kk` which—although it doesn't look that way—has the same scope as the local variables. The global version of `jj` remains in existence until the SUPERCODE execution ends.

One might be tempted to wonder if these scoping rules are really necessary. Well, in big programs they are. The only alternative is to ban name conflicts. In other words, if there was a global variable named `i`, one could not create local variables with the same name. In a multi-programmer project there is no way to ensure that such conflicts do not routinely cause big problems. The scope rules are relatively easy to live with after some practice.

Jump Statements

Jump statements unconditionally transfer control. The shell understands three of these: `break` statements, `continue` statements, and `return` statements.

The `break` statement can occur only inside loops. It causes immediate termination of the enclosing loop. The `continue` statement also can occur only inside loops. It causes the next iteration of the loop to start immediately.

The `return` statement can only appear inside functions. A return statement without an expression can only appear inside a function returning type `Void` (i.e., a subroutine). It causes control to return to the statement after the function call. Hitting the terminating curly bracket of a function is equivalent to executing a return statement without an expression. Again, this should only happen in a subroutine.

A return statement with an expression can be used only in functions returning values. This value is returned to the caller. If required, the value is converted to the return type of the function. To see how `return` statements work consider a function for calculating factorials

```
In[1]:
Integer factorial(Integer n)
{
  return (n > 1) ?  n * factorial(n - 1) :  1;
}
In[5]:
factorial(4);
  24
In[6]:
```

```
factorial(6);
  720
In[6]:
```

Notice that shell functions can be recursive!

It is also possible to return references:

```
In[1]:
Real x = 0;
In[2]:
Real &reftest()
{
  return x;
}
In[6]:
reftest() = 3.14159;
In[7]:
x;
  3.14159
In[8]:
```

This is certainly an interesting way to assign a value to `x`.

2.4 Classes

The SUPERCODE shell is able to manipulate sophisticated entities called *objects*. Due to the recent interest in a methodology called object-oriented programming, a number of definitions of *object* are available in the literature. We use the one by Booch [7]: "An object has state, exhibits some well-defined behavior, and has a unique identity." A simple example of an object is a variable `c` holding a complex number. The variable has a state ($c = 3 + 4i$, for instance), it has behavior governed by the rules of complex arithmetic, and it has a unique identity (`c` is different than another complex variable `d`).

An important concept closely related to that of an object is a *class*. A class is a template that describes the structure and behavior of a set of objects. A common synonym for class is `type`. The rest of this section will deal with the description of classes that the SUPERCODE supports.

2.4.1 Complex numbers

The class description for complex numbers as currently implemented by the SuperCode shell is

```
class Complex {
public:
  Complex();
  Complex(Real r);
  Complex(Real r, Real i);
  Complex(const Complex &c);
  ~Complex();

  Complex &operator=(const Complex &c);
  friend Complex operator+(Complex lc, Complex rc);
  friend Complex operator-(Complex lc, Complex rc);
  Complex operator-(Complex c);
  friend Complex operator*(Complex lc, Complex rc);
  friend Complex operator/(Complex lc, Complex rc);

  friend Fstream &operator<<(Fstream &os, Complex x);
  friend Fstream &operator>>(Fstream &os, Complex &x);
  friend Ostream &operator<<(Ostream &os, Complex x);

  Complex conj();

  Real re, im;
};
```

FORTRAN programmers will probably find the above notation to be strange. This notation is borrowed from the C++ programming language. For more details, see any of the C++ references.

A class is made up of *members* and *friends*. The `Complex` class above has two data members (the real part `re` and the imaginary part `im`), seven function members (four versions of the `Complex` function, `~Complex`, `operator=`, and `conj`), and the seven friend functions (`operator+`, etc).

The first four member functions of the `Complex` class are called *constructors*. They allow one to declare complex number is various ways. For instance,

```
Complex x1; // x is initialized to 0 + 0i
```

```
Complex x2(-12.32); // x2 is initialized to -12.32 + 0i
Complex x3(2.718, -3.142); // x3 is initialized to 2.718 - 3.142i
Complex x4(x3); // x4 is initialized to the value of x3
```

Constructors are usually called automatically by the shell. In the above example the `x1` declaration generates a call to the first constructor in the class description, the `x2` declaration generates a call to the second, the `x3` declaration generates a call to the third, and the `x4` declaration generates a call to the fourth. Like `Real` and `Integer` variables, it is also possible to alternatively use an assignment syntax to generate calls to the constructors

```
Complex x5 = 23.5; // x5 is initialized to 23.5 + 0i
Complex x6 = Complex(1, 2); // x6 is initialized to 1 + 2i
```

The declaration of `x5` generates a call to the constructor taking one `Real` argument while the declaration of `x6` generates a call to the constructor taking the `Complex` reference argument.

The ∼`Complex` member function is called a `destructor`. Destructors are never called by the user, they are automatically called by the shell whenever a class object goes out of scope. Destructors generally are in charge of discarding memory associated with the object.

The functions with names of the form `operator`*op* implement simple complex number arithmetic. One does not need to call these functions explicitly. Rather, when the shell sees an operation involving `Complex` quantities it picks out the right version to call. For example, one can execute the following expressions

In[1]:
`Complex a(1.2, 2.3), b(-1), c(6.1, -7.8);`
In[2]:
`-a * (b + c);`
  (2.718, -3.142)
In[3]:

Eventually, the shell will support a complete set of arithmetic functions involving `Complex` variables. However, the simple operations described in the class description are currently the only ones allowed.

Unlike `operator=` etc, some member functions must be called explicitly. The syntax for accessing these member functions is

object-name . *function-member-name*(*argument-list*)

For instance, the `conj` member function (which returns the complex conjugate) is called as follows

In[1]:
`Complex a(1.2, 2.3);`
In[2]:
`a.conj();`
  (1.2, -2.3)
In[3]:

Data members are accessed similarly:

object-name . *data-member-name*

Hence,

In[1]:
`Complex a(1.2, 2.3);`
In[2]:
`a.re; a.im;`
  1.2
  2.3
In[3]:

The remaining friend functions `operator<<` allow `Complex` quantities to use the stream I/O facility to be described in Section~2.5. Finally, note that the shell pre-defines six `Complex` work variables (`_ca`, `_cb`, `_cc`, `_cd`, `_ce`, `_cf`) that can be used in expressions without the need to declare them.

2.4.2 3-D Arrays

The three-dimensional array class description is

```
class TYPEArray3 {
public:
  TYPEArray3();
  TYPEArray3(Integer dim1, Integer dim2, Integer dim3);
  TYPEArray3(const TYPEArray3 &arr);
  ~TYPEArray3();

  TYPEArray3 &operator=(const TYPEArray3 &mat);
```

```
    friend TYPEArray3 operator+(const TYPEArray3 &la, const TYPEArray3 &ra);
    friend TYPEArray3 operator-(const TYPEArray3 &la, const TYPEArray3 &ra);
    friend TYPEArray3 operator*(const TYPEArray3 &la, const TYPEArray3 &ra);
    friend TYPEArray3 operator/(const TYPEArray3 &la, const TYPEArray3 &ra);

    friend Fstream &operator<<(Fstream &os, const TYPEArray3 &arr);
    friend Ostream &operator<<(Ostream &os, const TYPEArray3 &arr);

    TYPE &operator()(Integer i, Integer j, Integer k);
    TYPE &data();

    Integer dim1();
    Integer dim2();
    Integer dim3();
};
```

where `TYPE` = `Complex`, `Integer`, or `Real` gives the type of the elements contained in the Array3.

The Array3 constructors allow the user to specify the initial size of these objects:

```
IntegerArray3 A1; // A1 initially is 0 × 0 × 0
RealArray3 A2(3, 2, 2); // A2 initially is 3 × 2 × 2
RealVector A3 = A2; // A3 initially has size/elements of A2
```

Unless initialized to another object, three-dimensional arrays are initially filled with zeros. The `operator()` member function allows Array3 elements to accessed using a FORTRAN-like syntax

In[1]:
`A2(3, 1, 2) = -14.5; A2(2, 2, 1) = 9;`
In[2]:

Furthermore, like FORTRAN, Array3 are indexed from 1. Array3 types are printed by the shell using a format compatible with the *Mathematica* symbolic algebra program

In[1]:
`A2;`
{
{{ **0, 0** },

```
{ 0, 9 },
 { 0, 0 }},
{{ 0, 0 },
 { 0, 0 },
 { -14.5, 0 }}
}
In[2]:
```

Eventually, the shell will support a complete set of arithmetic functions involving Array3 variables. However, currently, the limited set of operations described above is all that has been implemented. Note that `operator*` and `operator/` do elementwise multiplication and division.

The `data` member function is used to pass the location of a Array3's storage to a routine that cannot deal with the composite Array3 structure (for example, a FORTRAN library function). Array3 data is stored contiguously according to FORTRAN conventions. The `dim1`, `dim2`, and `dim3` member functions return the dimensions of the Array3.

2.4.3 Matrices

The matrix class description is

```
class TYPEMatrix {
public:
  TYPEMatrix();
  TYPEMatrix(Integer nrows, Integer ncols);
  TYPEMatrix(const TYPEMatrix &mat);
  ~TYPEMatrix();

  TYPEMatrix &operator=(const TYPEMatrix &mat);
  TYPEMatrix &operator=(TYPE s);
  friend TYPEMatrix operator+(const TYPEMatrix &lm, const TYPEMatrix &rm);
  friend TYPEMatrix operator-(const TYPEMatrix &lm, const TYPEMatrix &rm);
  friend TYPEMatrix operator*(const TYPEMatrix &lm, const TYPEMatrix &rm);
  friend TYPEMatrix operator*(TYPE s, const TYPEMatrix &mat);
  friend TYPEMatrix operator*(const TYPEMatrix &mat, TYPE s);
  friend TYPEMatrix operator%(const TYPEMatrix &lm, const TYPEMatrix &rm);
  friend TYPEMatrix operator/(const TYPEMatrix &lm, const TYPEMatrix &rm);
  friend TYPEMatrix operator/(TYPE s, const TYPEMatrix &mat);
```

```
    friend TYPEMatrix operator/(const TYPEMatrix &mat, TYPE s);

    friend TYPEMatrix operator&(const TYPEMatrix &lm, const TYPEMatrix &rm);

    friend Fstream &operator<<(Fstream &os, const TYPEMatrix &mat);
    friend Ostream &operator<<(Ostream &os, const TYPEMatrix &mat);

    TYPE &operator()(Integer i, Integer j);
    TYPE &data();

    Integer rows();
    Integer cols();
  };
```

where TYPE = Complex, Integer, or Real gives the type of the elements contained in the matrix.

The matrix constructors allow the user to specify the initial size of these objects:

```
IntegerMatrix A1; // A1 initially has 0 rows, 0 cols
RealMatrix A2(3, 2); // A2 initially has 3 rows, 2 cols
RealVector A3 = A2; // A3 initially has size/elements of A2
```

Unless initialized to another object, matrices are initially filled with zeros.

The `operator=` functions allows one to assign matrices and scalars to matrices. If `a` and `b` are matrices, the expression `a = b` replaces the contents of matrix `a` with that of `b`. For this operation to work, `a` and `b` must be the same size. Assigning a scalar to a matrix (as in `a = 3.14`) sets all of the elements in the matrix to the scalar value.

The shell supports a fairly complete set of arithmetic functions involving matrix variables. Note that `operator*` and `operator/` do elementwise multiplication and division while `operator%` does matrix multiplication. For example,

In[1]:
```
RealMatrix a(2,2), b(2,2);
```
In[2]:
```
a(1,1) = 1; a(1,2) = 2; a(2,1) = 3; a(2,2) = 4;
```
In[3]:
```
b(1,1) = -1; b(1,2) = -3; b(2,1) = 3; b(2,2) = 4;
```

In[4]:
```
a * b;
```
{{ **-1, -6** },
{ **9, 16** }}

In[5]:
```
a % b;
```
{{ **5, 5** },
{ **9, 7** }}

In[6]:

Scalar multiplication is also supported.

The function `operator&` is fairly unique in that it allows matrix concatenation. For example,

In[1]:
```
RealMatrix a(2,2), b(2,2);
```
In[2]:
```
a(1,1) = 1; a(1,2) = 2; a(2,1) = 3; a(2,2) = 4;
```
In[3]:
```
b(1,1) = -1; b(1,2) = -3; b(2,1) = 3; b(2,2) = 4;
```
In[4]:
```
a & b;
```
{{ **-1, 2, 3, 4** },
{ **-1, -3, 3, 4**}}

For this operation to work, all the matrices involved must have the same number of rows.

It is possible to build up complicated expressions involving matrices and scalars. For example:

In[1]:
```
RealMatrix a(2,1), b(2,1), c(2,2), d(2,2);
```
In[2]:
```
d = (a & b) + 3.0 * c;
```

The `operator()` member function allows matrix elements to accessed using a FORTRAN-like syntax

In[1]:
```
A2(3, 1) = -14.5; A2(2, 2) = 9;
```
In[2]:

Furthermore, like FORTRAN, shell matrices are indexed from 1. Matrix types are printed by the shell using a format compatible with the *Mathematica* symbolic algebra program

```
In[1]:
A2;
  {{ 0, 0 },
   { 0, 9 },
   { -14.5, 0 }}
In[2]:
```

The `data` member function is used to pass the location of a matrix's storage to a routine that cannot deal with the composite matrix structure (for example, a FORTRAN library function). Matrix data is stored contiguously by columns to be compatible with FORTRAN. The `rows` and `cols` member functions return the dimensions of the matrix.

2.4.4 Vectors

The vector class description is

```
class TYPEVector : public TYPEMatrix {
public:
  TYPEVector();
  TYPEVector(Integer nrows);
  TYPEVector(const TYPEVector &vec);
  ~TYPEVector();

  TYPEVector &operator=(const TYPEVector &vec);
  TYPEVector &operator=(TYPE s);

  TYPE &operator()(Integer i);
  TYPE &data();

  Integer length();
};
```

where `TYPE` = `Complex`, `Integer`, or `Real` gives the type of the elements contained in the vector.

The `TYPEVector` class is said to *inherit* from the `TYPEMatrix` class. This means that any routine that expects to be passed a `TYPEVector` can be passed a `TYPEVector`. (Note: the inverse is not true: one cannot pass a `TYPEMatrix` to a routine expecting a `TYPEVector`.) This feature means that all of the binary operations involving matrices are automatically supported for vector. Moreover, it is possible to mix matrices and vectors in expressions.

The vector constructors allow the user to specify the initial size of these objects:

```
IntegerVector A1; // A1 initially has 0 rows
RealVector A2(3); // A2 initially has 3 rows
RealVector A3 = A2; // A3 initially has size/elements of A2
```

Unless initialized to another object, vectors are initially filled with zeros. The `operator()` member function allows vector elements to accessed using a FORTRAN-like syntax

```
In[1]:
A2(3) = -14.5; A2(2) = 9;
In[2]:
```

Furthermore, like matrices, shell vectors are indexed from 1. Vector types are printed by the shell using a format compatible with the *Mathematica* symbolic algebra program

```
In[1]:
A2;
  {{ 0 },
   { 9 },
   { -14.5 }}
In[2]:
```

The `data` member function is used to pass the location of a vector's storage to a routine that cannot deal with the composite vector structure (for example, a FORTRAN library function). Vector data is stored contiguously to be compatible with FORTRAN. The `length` member function returns the dimensions of the vector.

2.4.5 Strings

`String` variables represent character string data. Their current class description is

```
class String {
public:
  String();
  String(const String &str);
  ~String();

  String &operator=(const String &str);
  friend String operator+(const String &ls, const String &rs);

  friend Fstream &operator<<(Fstream &os, const String &str);
  friend Ostream &operator<<(Ostream &os, const String &str);
};
```

The string constructors allow users to declare `String` objects as follows

```
String a, b(``hello''), c = ``goodbye'';
```

In this example, `a` is initially the null string “”, `b` is initially `` `hello'' `` and `c` is `` `goodbye'' ``. Other than stream I/O, the only operation currently supported for strings is concatenation (denoted by the '+' operator). For instance,

```
a = b + `` '' + c;
```

would result in `a` containing “`hello goodbye`”.

2.5 Shell Input and Output

As we have seen, the simplest way to get information regarding a variable is to use it in an expression statement

In[1]:
```
Real x = 46.89;
```
In[2]:
```
x;
```
  48.89
In[3]:

The shell tries to be somewhat selective about the type of expressions that it outputs expressions for. In particular, the shell will not output a result when the first operator encountered is an assignment operator:

In[1]:
```
Real y;
```
In[2]:
```
y = 1.2 + 3.4;
```
In[3]:
```
y;
```
  4.6

In[4]:

This conflicts slightly with the C++ convention that assignment is just another operator but cuts down on generally unwanted output.

It is also possible to get output by using the `Ostream` class. `Ostreams` currently cannot be instantiated[3] for in the SUPERCODE shell but a pre-defined variable `cout` has been provided. The `cout` variable is used according to C++-style stream I/O conventions [4, 5, 6]. For example, to output two numbers to the terminal one simply types

In[1]:
```
Real a = 1.4, b = -1.3;
```
In[2]:
```
cout << a << `` '' << b << ``\n'';
```
  1.4 -1.3

In[3]:

As this example demonstrates, it is possible to output various types of variables using a single statement. In particular, notice the string containing the space: it is necessary to ensure that the two numbers don't run together. It is possible to output complex numbers, matrices, strings, and vectors in addition to the basic types. One can embed carriage returns in the output using the '\n' character

In[1]:
```
Real a = 1.4, b = -1.3;
```
In[2]:
```
cout << ``The Answer is \n'' << a << `` '' << b;
```

[3]Instantiation is a synonym for creation.

```
  The Answer is
  1.4 -1.3
In[3]:
```

Consistent with C++, a carriage return is *not* automatically placed after each stream output statement involving `cout`. This means that executing

```
for (i = 1; i <= 5; i++)
  cout << i;
```

produces

```
  12345
```

The `cout` stream is meant for output. Accordingly, it is buffered in 1024 character blocks. Another stream, `cerr`, which is line buffered is also usable in the shell. This stream is useful for error messages.

It is also possible to perform stream I/O with files. This is accomplished using the `Fstream` and `ios` classes

```
class Fstream {
public:
  Fstream();
  Fstream(const String &str, Integer ioflag);
  ~Fstream();

  Void open(const String &str, Integer ioflag);
  Void close();
};

class ios {
public:
  static Integer in;
  static Integer out;
};
```

Notice that the `ios` class contains a new type of `static` data member. Static data members do not belong to a particular object. Rather, all instantiations of the class access the same value. Since the static member is not tied to a particular object they need not be accessed using the *object.member* syntax. Instead, they can be referred to using a *qualified name* with the syntax *class-name* :: *static-data-member-name*

Therefore, to access the static member `out` in class `ios` one can type `ios::out`. Note that it *would be* possible to use the usual *object.member* syntax if an `ios` object could be instantiated. However, since the `ios` class has no constructors that is not possible in the SUPERCODE shell. In this case, the qualified name must be used. Note that it is possible to have static member functions as well. They can be called using the qualified name syntax

class-name :: *static-function-member-name*(*argument-list*)

The `Fstream` constructors allow the following shell code to be executed. This fragment

In[1]:
```
Real a = 1.4, b = -1.3;
```
In[2]:
```
Fstream fio(``test.dat'', ios::out);
```
In[3]:
```
fio << a << `` '' << b << ``\n'';
```
In[4]:

would result in a file called "test.dat" containing the text data

```
1.4 -1.3
```

The declaration of `fio` creates (if necessary) the file "test.out", cleans out all of the current contents (if necessary), and opens the file for writing. The stream output statement actually store the data in the file. Additional statements involving `fio` would append data to the file. The file remains open until the file closing member function

```
fio.close();
```

is issued.

It is possible to re-open a file using the file opening member function

```
fio.open(``test.dat'', ios::in);
```

The `open` member function takes the same arguments as the `Fstream` constructor. Notice that the above statement opens `test.dat` for *reading*. It is now possible to re-read the contents of the file by "re-directing" the stream operators:

In[1]:
```
Real a1 = 0.0, b1 = 0.0;
```
In[2]:
```
fio >> a1 >> b1;
```
In[3]:
```
a1; b1;
```
1.4
-1.3
In[4]:

Currently, it is not possible to read 3-D array, matrix, string, or vector variables.

The shell supports a wide variety of formatting options. See a C++ book for details.

Chapter 3

Writing Modules

3.1 Introduction

The computational part of the SUPERCODE is divided into a number of *modules*. Each of these modules contains data and functions that are related by a single unifying abstraction. For instance, a module named *Const* might contain the values of useful mathematical and physical constants. Or, perhaps, a module named *TFCoil* might contain data and functions relevant to calculating characteristics of the ITER toroidal field system.

There are two main reasons for the division into modules. First, this partitioning helps reduce the complexity inherent in a computer program the size of the SUPERCODE. We expect that, eventually, the SUPERCODE will consist of hundreds of thousands of statements. Most people will not have the time, or the inclination, to understand all of the code. Modules allow people to consider the code in small chunks that can reasonably be understood. The second reason for creating modules is practical. Several physicists and engineers from various institutions will be simultaneously working on the SUPERCODE project. Modules allow parts of the code to be designed and revised independently.

The act of partitioning the code does not guarantee that we will succeed in our two goals. For instance, suppose that a user of the *TFCoil* module made explicit assumptions regarding the details of the algorithm used to compute TF coil stresses. Then, changes to that algorithm would require changes elsewhere in the code. This coupling, which can be quite subtle, serves to increase complexity and disallow independent development.

It is important to recognize that we are not stating that all dependencies between modules are undesirable. Clearly, some coupling between modules in unavoidable. However, the *type* of coupling is the issue. It is not unreasonable to expect that another engineering module might be concerned with the values of the TF coil stresses. However, other modules should not care whether the TF stresses are computed using a simple scaling algorithm or an elaborate 3D solve.

In order to figure out how to intelligently handle the coupling between modules, it is useful to think of a module as having two parts: an *interface* and an *implementation*. The interface of a module describes how it appears to an outside observer or, put another way, how it behaves. The implementation of a module specifies the mechanisms that achieve the desired behavior. Consider the TF coil module. The interface consists of the data needed to perform its calculations (e.g., the shape of the TF coil, the magnetic field, etc), the functions one needs to call to do the calculations (e.g., `calcStresses`), and the results from the calculations (e.g., the stresses, nuclear heating, etc). The implementation consists of the specific algorithms used to calculate the results.

Eliminating the dependencies we are worried about is achieved by *encapsulating* the details of a module's implementation. This, in turn, can be accomplished by making sure that users always use function calls to calculate things. Here, users of the *TFCoil* module would call a routine `calcStresses` to get the TF coils stresses. As we mentioned, this function might compute the stresses using a simple scaling method or an elaborate 3D solve. However, as long as the interface to this calculation (i.e., the name of the function) doesn't change, the source for modules that use the routine need not be changed.

Function calls provides a way to encapsulate module implementations. However, this alone is not sufficient. First, modules will invariably need to store data in order to perform their calculations. Furthermore, some of this data will be peculiar to a specific algorithm. If other modules access this data, the undesirable coupling returns. In addition to data, module writers will probably want to define functions to implement algorithm-specific calculations. What is to stop users from calling these routines?

It seems that there needs to be two classes of information in a module: *public* data and functions that are supposed to be accessed by users of the module and *private* data and functions that are for internal use only. In the *TFCoil* module, an example of public data might be the desired toroidal field at the plasma center (an input) or the stress in the inner TF leg (an output). The routine `calcStresses` would be a public function callable by users of the module. Coefficients used for scaling calculations might be private data members while a routine that performs part of the stress calculation might be a private function.

In this section, we have introduced the concept of modules and the idea of having public and private data and function module members. Up to this point the discussion has been quite general. In the balance of this chapter we will provide specifics as to how to write such modules for incorporation into the SuperCode. We will also describe how to access the data and routines making up a module from the SuperCode shell.

3.2 Writing the Interface

The first step in developing a module is creating the interface. This means determining what the public data and functions are and what the private data and functions are. Once this is done, it is necessary to pass this information on to other module writers since they cannot use a module unless *they* know what the interface looks like.

For the SuperCode project, we have chosen to formalize this process of exchanging information by requiring that all developers supply a *module description file* specifying the interface for each module they contribute. This requirement has a number of desirable features

1. *Reliability.* Since module description files have a particular format (soon to be described) they can be read by a computer program which then generates the necessary code and common block declarations required to allow modules to talk to each other. This program also can generate the code required to give the shell access to the module members. This automatic generation of interface code reduces programming errors and makes handling changes in the interface less tedious for the user.

2. *Language-independence.* The required interface code depends on the programming language. Since this code is generated automatically, it is possible to have some modules written in FORTRAN and some modules written in C++ while, at the same time, hiding this detail from the user.

3. *Portability.* The required interface code may depend on the computing platform. Since this code is generated automatically these system-specific issues can be hidden from the user.

4. *Documentation.* A properly-written module description file can serve as documentation for a module. Therefore, no secondary documents need be written.

We believe these features outweigh the need to learn how to correctly format the module description file.

3.2.1 Module Description File Naming Conventions

By convention, module description files are named *ModName.mod* where *ModName* is a capitalized one-word label for the module. For example, the module description file name for the *TFCoil* module would be *TFCoil.mod*.

3.2.2 Module Description File Format

A module description file has the basic syntax

```
module module-name-with-info {
  module-element
  module-element
     ⋮
  module-element
}
```

where the *module-name-with-info* specifies information about the module as a whole and *module-element* is one of

- The access specifier "`public:`".
- The access specifier "`private:`".
- A data member declaration.
- A function member declaration.
- A module import declaration. (This tells what modules this module uses data or functions from.)

- An exception declaration. (This describes a warning, error, or fatal that this module can generate.)

There may be any number of *module-element*s and they may be listed in any order. Also, comments of the type discussed in Section 2.3.2 can be used anywhere.

A concrete example of these elements is contained in the module description file below which gives the interface for the hypothetical module *Example*:

```
// Example.mod

module Example - ``A sample module'' {
  Uses Const, Sys;
  Errors badGridSize
                      - ``Illegal number of grid points.''
                      - ``Grid size < 0 was passed to resizeGrid.'';
public:
  fortran Void ctor()
                      - ``Constructor for the Example module.''
                      - ``Initializes grid and allocates storage.'';
  fortran Void dtor()
                      - ``Destructor for the Example module.'';

  fortran Void resizeGrid(const Integer &newNGPTS)
                      - ``Resets the grid to the specified size.'';

  fortran Integer ngpts()
                      - ``Returns the current number of grid points.'';
  RealVector gridVals
                      - ``Grid location values.''
                      - ``By default, the grid is rectangular.''
                      - ``This variable is for reading ONLY.'';

private:
  Integer cngpts
                      - ``The current number of grid points.'';
  fortran Void calcGrid(RealVector &gvals)
                      - ``Computes grid location values.'';
}
```

This module description file declares four public functions (`ctor`, `dtor`, `ngpts`, and `resizeGrid`), one public data member (`gridVals`), one private function (`calcGrid`), and one private data member (`cngpts`). It also indicates that is module uses data or functions from the `Sys` and `Const` modules. Finally, an error exception `badGridSize` is declared. We will now examine the syntactic elements of the file in more detail.

Module Name Specifier

Line 3 of the sample file shows an example of the *module-name-with-info* element:

```
Example - ``A sample module.''
```

The first word specifies the name of the module which should match the first part of the file name. The next part is a construct that gives information that is to be stored in the driver-shell's run-time database. In this case, the string "A sample module." is associated with the name "Example". Eventually, it will be possible to access this information using shell commands but that capability has not been implemented yet. As the sample file shows, this pattern of the name of an object followed by an information string (or strings) appears many times. In each case, the string(s) represent descriptive information about the object that is to be stored in the shell database.

As the sample file shows, it is possible to associate more than one line of information with an object. This is accomplished by simply appending more strings, separated by dashes, to the name. For example, a more detailed description of the module could be written

```
Example - ``A sample module.''
        - ``This module manages a one-dimensional rectangular grid array.''
```

By convention, the first string gives a brief description of the element. Subsequent strings should give more detailed information. In addition to supplying the information to the shell database, the information strings should also provide documentation for users of the module.

Access Specifiers

The access specifier "`public:`" indicates that subsequent function and data members are public. Accordingly, these members can be used by other modules. This remains the case until a "`private:`" access specifier is seen. At that point, all members are deemed private and they *cannot* be accessed by other modules. There can be any number of groups of public or private members; however, by convention, public members are listed first.

Data Member Declarations

As the sample file indicates, data members are declared in a manner very similar to variables in the shell (see Section 2.3.3). However, for compatibility with FORTRAN, the types of variables that can be declared is limited. In particular,

- The only allowed types are `Complex`, `Integer`, `Real`, `ComplexArray3`, `IntegerArray3`, `RealArray3`, `ComplexMatrix`, `IntegerMatrix`, `RealMatrix`, `ComplexVector`, `IntegerVector`, `RealVector`, `String`, and `Subroutine`.
- No constant or reference variables are allowed.
- Variable names must start with a letter.
- Variables cannot be initialized in the module description file.

Note that data members can be initialized using a mechanism to be described shortly. Moreover, in the future, the prohibition on initializing data members in the module description file may be relaxed. By convention, data member names should start with a lower case letter.

It is possible to declare several data members at once. For example

```
Integer i
          - ``Variable i description'',
        j
          - ``Variable j description 1''
          - ``Variable j description 2'',
        k,
        l
          - ``Variable l description'';
```

Note carefully the placement of commas and the semicolon. The system is relatively easy once you get the hang of it.

Function Member Declarations

As with data members, the function member declaration syntax is very similar to that of the shell language. Again, for compatibility with FORTRAN, there are limitations on the sorts of functions that can be declared. These are

- Functions can only return types `Integer`, `Real`, and `Void`.
- The return type cannot be a reference or a constant.
- If the function is actually implemented in FORTRAN, the `fortran` keyword must be placed in front of the declaration. By default, it is assumed that functions are written in C++.
- The only allowed argument types are `Complex`, `Integer`, `Real`, `ComplexArray3`, `IntegerArray3`, `RealArray3`, `ComplexMatrix`, `IntegerMatrix`, `RealMatrix`, `ComplexVector`, `IntegerVector`, `RealVector`, `String`, and `Subroutine`.
- All arguments, other than those of type `Subroutine` must be declared as references. It is possible to pass constant references as arguments (i.e., `const Real &`).
- Function names must start with a letter.
- There cannot be two members with the same name.

The sample module description file gives a number of examples of function member declarations. By convention, function member names should start with a lower case letter.

Module Import Declarations

The modules that the current module expects to get data or call functions from must be listed in a module imports declaration. The form of this declaration depends on whether the module is written in FORTRAN or C++. FORTRAN modules use the keyword `Uses` while C++ modules use the keyword `Inherits`. By convention, the module imports declaration is the first module element.

Exception Declarations

Since the SUPERCODE is interactive, it is important for modules to perform error checking. If a problem is found, the module can handle the error using an exception declared in the module description file. Three types of exceptions can be generated: warnings, errors, and fatals. These correspond to the exceptions described in Section 2.1.

An exception declaration essentially associates an exception name with a message. In the sample file, the exception `badGridSize` is associated with a relevant error message. Exception declarations have the same syntax as data member declarations except that the "type" is either `Warnings`, `Errors`, or `Fatals`. The information strings for exceptions are treated slightly differently though. In the case of exceptions, they represent the text of the message that is to be printed by the shell upon hearing about the exception. Accordingly, at least one information string for each exception must be supplied. Only the first three strings will actually appear in the error message but more can be provided in the module description file for documentation purposes. By convention, exception declarations are listed in order of increasing severity right after the module imports declaration. Also, exception names start with a lower case letter.

In a later section we will show how to actually raise exceptions.

Special Functions

Modules include two special function members called `ctor` and `dtor`. The `ctor` function is known as the *constructor* for the module and the `dtor` function is known as the `destructor` for the module. When the SUPERCODE is started up, the shell calls the `ctor` functions for all of the modules. When the `quit` directive is executed, the `dtor` functions are called in the reverse order.

Every module that stores information should define a `ctor` function in the module description file. Then, inside the `ctor` function, initialization functions should be performed with the goal of making sure that the module is in a consistent state at the start. Typical activities carried out in a `ctor` include initializing dimensions, resizing matrices and vectors, filling in data, or perhaps opening up files.

The use of the `dtor` function is much less common. However, if there are any clean-up activities that need to be performed when the code shuts down they should be done in the `dtor`. Storage is deallocated automatically so that is not a worry. However, closing or writing a file may be something a `dtor` might do.

Modules that do not need the services of a constructor or destructor need not define one: they will be generated automatically.

3.2.3 MGen

We alluded to the fact that the module description file could be read by a computer program which would produce the code needed to allow modules and the shell to communicate. The name of this program is *MGen* and it can be executed on our sample module description file by simply typing

```
MGen Example.mod
```

When *MGen* gets done reading the module description file, you will notice that four files have appeared in your directory:

- *Example.i* This file contains common blocks and definitions required to allow *Example*'s FORTRAN member functions to access the *Example* module's data and function members along with the public data and function members of the *Sys* and *Const* modules that *Example* imports.
- *Example.h* This file contains definitions required to allow *Example*'s C++ member functions and the shell to access *Example* module members.
- *ExampleMBase.m* This file contains FORTRAN glue code required to allow the shell to access *Example* module members.
- *ExampleCBase.cc* This file contains C++ glue code required to allow the shell to access *Example* module members.

Generally, module writers should not be concerned with the contents of these files. Moreover, these files contain some of the yuckiest looking code ever written so it's best to leave these alone.

3.3 Writing the Implementation

Having shown how to write a module interface, it is now necessary to show how to write the module implementation. The details of this procedure depend on the programming language. We will first show how to write an implementation in FORTRAN and then how to write one in C++. Note, however, that it is not necessary for a module to be written entirely in one language! Some functions can be written in FORTRAN for efficiency and some can be written in C++ for convenience.

3.3.1 Using FORTRAN

FORTRAN module implementations are initially read by a pre-processor called *MPPL*. MPPL converts the implementation to FORTRAN77 code which can then be compiled. Files that are designed to be read by MPPL must have the suffix `.m` while pure FORTRAN files have the suffix `.f`.

MPPL supports a number of enhancements to FORTRAN including

- Free format input.
- Shell-style `//` comments.
- Sophisticated macro capabilities.
- Simplified looping and selection statements.

These capabilities are described in detail in the MPPL manual [8]. Note that module writers do not *have* to use these extensions: with a few exceptions, modules can be written in pure FORTRAN77. However, MPPL's enhancements make FORTRAN a much more pleasant language to use.

In the balance of this section, we will show, by means of our sample module *Example*, how to write a FORTRAN implementation. For simplicity, we assume that all of the source code for the module resides in the file *Example.m*.

Near the top of *Example.m* (before any functions or subroutines) it is necessary to have the line

```
include Example.i
```

Note that there are no quotes around the file name. This directive causes the contents of the *Example.i* definitions file generated by *MGen* to be read.

Now we can begin writing the member functions. Let us start with the constructor. It is written

```
subroutine Example_ctor
BeginExampleMember()

  call resizeGrid(8)
  return

EndMember
```

Part of this looks like standard FORTRAN and part doesn't. The first difference is the name of the subroutine. The "`Example_`" is prepended to `ctor` here to indicate that this is a member of the module *Example*. The next line is a macro that causes the common blocks holding the data members of the *Example* module to be included. All calls to the `BeginExampleMember`[1] macro must be balanced by a call to the `EndMember` macro which replaces the subroutine's `end` statement. In between these macro calls is fairly standard FORTRAN code that initializes the grid to a size of 8.

At this point it is pertinent to discuss references to module members. For the most part, it is possible to refer these members using the name as written in the module description file (e.g. `resizeGrid`). The only exception to this rule is when a public member of an imported module contains a member with the same name as a member of the present module. An example of this situation is the `ctor` function which is a member of all modules. If one simply included a line

```
call ctor
```

in a member function, it is not clear whether we are supposed to use the `ctor` in *Sys*, *Const*, or *Example*. Therefore, MPPL will generate an error saying that the call is ambiguous. The correct way to do it is to prepend the module name:

```
call Sys_ctor
```

This say that we are calling *Sys*'s `ctor`. Note that all members can be referred to in this way (e.g., `Example_resizeGrid`, `Example_cngpts`, etc). However, prepending the module name is not necessary unless the reference would otherwise be ambiguous.

Having written the constructor, we move to the destructor

[1]Note: if the module name is `Mod`. the macro name is `BeginModMember`.

```
subroutine Example_dtor

  return

end
```

Since this routine doesn't do anything, we could have omitted its declaration from the module description file. Then it would have been written by *MGen*. Notice that there is no `BeginExampleMember/EndMember` macro pair for this subroutine. This is not necessary because the routine does not access any of the module's members. Leaving off these unnecessary macro calls will decrease compilation time and code size so, whenever possible, they should be eliminated.

The `resizeGrid` member shows how to handle arguments

```
subroutine Example_resizeGrid(newNGPTS: integer)
BeginExampleMember(Sys)

  if (newNGPTS < 0) then
    call except(badGridSize)
  endif
  cngpts = newNGPTS
  call rvResize(RealVector(gridVals), newNGPTS)
  call calcGrid(RealVector(gridVals))

  return

EndMember
```

Notice that the type of the argument is declared using the syntax

arg-name : *type*

where the *type* maps to the corresponding type from the shell or module description file according to Table 3.3.1. No `const` keywords appear here since FORTRAN has no mechanism for ensuring that `const` items do not get changed. It is up to the programmer to ensure that these are read-only. Also, it is not necessary to place the `&` indicating that references are passed since this is always the case with FORTRAN. In addition, see that the `BeginExampleMember` macro takes an argument "Sys". This argument

Table 3.1: Shell/Module Description File Type Mapping to MPPL Types

Shell/MDF Type	MPPL Type
`Complex`	`complex`
`Integer`	`integer`
`Real`	`real`
`ComplexArray3`	`complexarray3`
`IntegerArray3`	`integerarray3`
`RealArray3`	`realarray3`
`ComplexMatrix`	`complexmatrix`
`IntegerMatrix`	`integermatrix`
`RealMatrix`	`realmatrix`
`ComplexVector`	`complexvector`
`IntegerVector`	`integervector`
`RealVector`	`realvector`
`Subroutine`	`subroutine`

specifies that the routine will make use of variables and/or functions from the imported `Sys` module. If the routine also made use of variables from the `Const` module the macro would read `BeginExampleMember(Const,Sys)`.

The `resizeGrid` member is supposed handle changes in the grid size. First, it checks the desired grid size to make sure it is positive. If it isn't, it calls the exception raising subroutine `except`. This subroutine expects one argument: the label of the exception. If the exception is a warning, an message is printed and execution proceeds on the line following the `except` call. Errors and fatals, on the other hand, do not return. If the new grid size is acceptable, the private variable `cngpts` is updated and the routine `rvResize` (a member of the *Sys* module, see *Sys.mod*) is called. This routine, which has the declaration,

```
Void rvResize(RealVector &vec, const Integer &nm)
```

resizes `gridVals` to have `newNGPTS` elements. Note that the "`RealVector(···)`" is required. This is a macro call that ensures that all of the correct information is passed to a function expecting a `RealVector &` as an argument. There are corresponding macro calls for `ComplexMatrix`, `ComplexVector`, etc. Be sure to use this macro or else disaster will follow since FORTRAN does no type-checking. Finally, the private member function `calcGrid` is called to actually compute the grid values and store them into `gridVals`.

To resize Array3 variables, use the routines `ca3Resize`, `ia3Resize`, and `ra3Resize`. These take three dimensions. For example,

```
call ca3Resize(ComplexArray3(x), 3, 5, 7)
```

resizes a `ComplexArray3` variable `x` to be $3 \times 5 \times 7$. Be sure to use the "`ComplexArray3(···)`" macro.

The member `ngpts` is a very simple function

```
integer function Example_ngpts()
BeginExampleMember()

  return(cngpts)

EndMember
```

Making `ngpts` a function means that users of the module can look at the number of grid points but cannot modify it haphazardly. This is the best approach if there needs to be side effects associated with changing a variable. Note that the syntax for returning answers from functions is different than standard FORTRAN. The normal approach of assigning to the function name will not work in SuperCode modules.

We finally come to the `calcGrid` routine which actually does some arithmetic

```
subroutine Example_calcGrid(gvals:  realvector)
BeginExampleMember(Const)
  integer i

  do i = 1, rows(gvals)
    gvals(i) = twoPi * float(i - 1) / float(rows(gvals))
  enddo

  return

EndMember
```

The constant `twoPi` comes from the *Const* module. This routine illustrates a few important points. First, all of the matrix and vector types know what their dimensions are. The number of rows of a vector[2] can be accessed using the `rows` macro while the number of rows or columns of a matrix can be accessed using the `rows` or `cols` macros. The second important point is that, while `RealVector` types must be handled specially when passing them as arguments, they can be manipulated in arithmetic expressions exactly like FORTRAN arrays.

Some of the syntax in this section probably seems strange. However, a little practice should make writing modules in FORTRAN a snap.

3.3.2 Using C++

Writing modules in FORTRAN requires the use of a number of MPPL extensions. This is because FORTRAN77 has little inherent support for modularity or information hiding.[3] C++, on the other hand, includes these elements by default. Moreover, you will see that the syntax is extremely close to that of the shell language.

In the balance of this section, we will rewrite our sample module in C++. The only changes one needs to make to the module description file is the deletion of the `fortran` directives in front of the function member declarations and the replacement of `Uses` with `Inherits` in the import declaration line.

For simplicity, we assume that all of the source code for the module resides in the file *Example.cc*[4].

Near the top of *Example.cc* (before any functions or subroutines) it is necessary to have the line

```
#include ``Example.h''
```

This directive causes the contents of the *Example.h* definitions file generated by *MGen* to be read.

Now we can begin writing the member functions. Let us start with the constructor. It is written

[2] Vectors are assumed to be column vectors.

[3] The new FORTRAN90 standard incorporates many of the features we have had to implement with MPPL including modules and public and private member functions.

[4] The suffix `.cc` is assumed for C++ files.

```
Void Example::ctor()
{
  resizeGrid(8);
}
```

The name of the module is connected to the name of the member with a double-colon `::`. This replaces the underscore of FORTRAN. There are no common blocks in C++ so no `BeginMember`/`EndMember` pair of macros is required. The body of the function consists of a call to `resizeGrid` which initializes the grid to a size of 8.

The ambiguity resolution rules for C++ are the same as for FORTRAN except that member references have the forms `Example::resizeGrid`, etc. Again, prepending the module name is not necessary unless the reference would otherwise be ambiguous.

As before, the destructor is empty

```
Void Example::dtor()
{
}
```

Since this routine doesn't do anything, we could have omitted its declaration from the module description file. Then it would have been written by *MGen*.

The `resizeGrid` member shows how to handle arguments

```
Void Example::resizeGrid(const Integer &newNGPTS)
{
  if (newNGPTS < 0)
    except(badGridSize);
  cngpts = newNGPTS;
  rvResize(gridVals, newNGPTS);
  calcGrid(gridVals);
}
```

Notice that the formal argument declarations in C++ match exactly the ones in the module description file. In addition, the "`RealVector(···)`" macro is not required in C++ since this language knows how to pass matrix and vector types.

The member `ngpts` is very straightforward

```
Integer Example::ngpts()
{
  return(cngpts);
}
```

as is `calcGrid`:

```
Void Example::calcGrid(RealVector &gvals)
{
  Integer i;

  for (i = 1; i <= rows(gvals); i++)
    gvals(i) = (twoPi * (i - 1)) / rows(gvals);
}
```

Again, it is important to note the the matrix and vector types work just like FORTRAN arrays (including indexing from 1). Moreover, the rows and columns can be accessed using `rows` and `cols`.

3.4 Recompiling the Code

The SUPERCODE compile and link process is controlled by a powerful UNIX utility called *imake*. *Imake* is a front-end for the well-known UNIX *make* program. The primary advantage of *imake* is that it provides a way to hide all of the system-dependent information required to build a program from the user. As a result, Imakefiles (the *imake* equivalent of a makefile) are extremely short, even for complicated programs like the SUPERCODE.

After writing the interface and implementation, the next step in adding a module to the SUPERCODE is revising the Imakefile. The SUPERCODE Imakefile is *SuperCode/SC/Imakefile*. We will now discuss how to modify this file to add FORTRAN and C++ modules.

3.4.1 Adding a Simple FORTRAN Module

The Imakefile changes required to add a module consisting of a module description file and a *single* MPPL source file is shown below

```
SetDefaultOptimization(-O3)
```

```
#ifdef InObjectCodeDir

SourceFileDependencies()
ModFileDependencies()

SimpleCCModuleTarget(Const)
SimpleCCModuleTarget(Sys)
SimpleMpplModuleTarget(Example)

all::
→  $(RM) modlist.h
→  EchoModuleEntry(Const)
→  EchoModuleEntry(Sys)
→  EchoModuleEntry(Example)

AllTarget(SuperCode)
DependTarget()

OBJS = SuperCode.o ModConst.o ModSys.o ModExample.o

SuperCode.o:  modlist.h

SuperCode :  $(OBJS)
→  $(CCC) -Bstatic -o SuperCode $(OBJS) $(PROJECT_STDLIBS)

clean::
→  $(RM) SuperCode

InstallProgram(SuperCode,$(PROJECT_BINDIR))

Example.o: Const.i Sys.i

#else

MakeInObjectCodeDir()

#endif
```

Originally, this Imakefile built the SUPERCODE using only the *Const* and the *Sys* modules. The modifications required to add our sample module *Example* are shown in **bold face**. The arrows (→) indicate that these lines must begin with a TAB. Basically, one must simply specify the name of module in a few places and supply a line giving the include files for imported modules (it is not necessary to include *Example.i* on the line).

3.4.2 Adding a Simple C++ Module

If a module consists of a module description file and a single C++ source file, the modifications to the Imakefile are almost identical to those required to add a FORTRAN module. If our module *Example* were written in C++, the Imakefile above would become

```
SetDefaultOptimization(-O3)

#ifdef InObjectCodeDir

SourceFileDependencies()
ModFileDependencies()

SimpleCCModuleTarget(Const)
SimpleCCModuleTarget(Sys)
SimpleCCModuleTarget(Example)

all::
→  $(RM) modlist.h
→  EchoModuleEntry(Const)
→  EchoModuleEntry(Sys)
→  EchoModuleEntry(Example)

AllTarget(SuperCode)
DependTarget()

OBJS = SuperCode.o ModConst.o ModSys.o ModExample.o

SuperCode.o:  modlist.h

SuperCode :  $(OBJS)
```

```
→  $(CCC) -Bstatic -o SuperCode $(OBJS) $(PROJECT_STDLIBS)

clean::
→  $(RM) SuperCode

InstallProgram(SuperCode,$(PROJECT_BINDIR))
```

Example.o: Const.h Sys.h

```
#else

MakeInObjectCodeDir()

#endif
```

where, as before **bold face** indicates the required changes.

3.4.3 Adding a more Complex Module

It is not necessary for the source for a module to reside in a single source file. If this is the case, a slightly different set of modifications to the Imakefile is required. Consider, for instance, a new module called *Optimize*. This module is a front-end for the VMCON constrained optimization package. It consists of a module description file *Optimize.mod*, a C++ source file *Optimize.cc*, and a pure FORTRAN source file *VMCON.f*. The Imakefile changes required to add *Optimize* to the SUPERCODE (which already includes the *Example* module) are shown below

```
SetDefaultOptimization(-O3)

#ifdef InObjectCodeDir

SourceFileDependencies()
ModFileDependencies()

SimpleCCModuleTarget(Const)
SimpleCCModuleTarget(Sys)
```
ModuleTarget(Optimize,Optimize.o VMCON.o)
```
SimpleCCModuleTarget(Example)
```

```
all::
→   $(RM) modlist.h
→   EchoModuleEntry(Const)
→   EchoModuleEntry(Sys)
→   EchoModuleEntry(Optimize)
→   EchoModuleEntry(Example)

AllTarget(SuperCode)
DependTarget()

OBJS = SuperCode.o ModConst.o ModSys.o ModOptimize.o ModExample.o

SuperCode.o:  modlist.h

SuperCode :  $(OBJS)
→   $(CCC) -Bstatic -o SuperCode $(OBJS) $(PROJECT_STDLIBS)

clean::
→   $(RM) SuperCode

InstallProgram(SuperCode,$(PROJECT_BINDIR))

Optimize.o: Optimize.h
Example.o:  Const.h Sys.h

#else

MakeInObjectCodeDir()

#endif
```

where, as before **bold face** indicates the required changes. Basically, one must simply specify the name of the module, the object files that make up the module, and the include files the objects depend on. Unlike the simple case, *all* include files must be specified (not just the imports).

3.4.4 Recompiling

Anytime the Imakefile is changed, it is necessary to issue the command

```
make Makefile Makefiles
```

This tells *imake* to reconfigure its files for the new module. To compile and link the whole code simply type

```
make
```

Don't worry about the details, *imake* knows what to do.

If an error during one of the compiles, note the name of the file that had the problem. Then look in the *SUN4* directory and you will see a bunch of files with the suffix `.err`. Open up the file that has the correct file prefix in order to see the error. Upon fixing the problem, simply type `make` again.

If the build finishes without problems, a brand-new version of the SUPERCODE should appear as *SuperCode/SC/SUN4*. It can be executed by typing `SuperCode` in the *SuperCode/SC* directory if you include `./SUN4` in your path (this is done in *cshrc.scode*.

3.4.5 Compiling with Debugging Information

By default, *imake* compiles optimized code. However, sometimes it is necessary to make use of a debugger such as *dbx*. In order to do this simple change the line

```
SetDefaultOptimization(-O3)
```

in the Imakefile in the *SC* directory to

```
SetDefaultOptimization(-g)
```

Then, type "`make Makefile Makefiles`". At this point, you must simply recompile the files you wish to debug to generate debugging information. To switch back to using optimized code, change the `SetDefaultOptimization` macro back. See the manual for whatever debugger you are using for debugging instructions.

3.5 Accessing Modules From the Shell

Data and function members of a module can be accessed from the shell. The safest way to do this is to prepend "`ModName::`" (where `ModName` is the name of the module) to the member names. Using this convention, we can access members of our sample module *Example* as follows

```
In[1]:
Example::ngpts();
  8
In[2]:
Example::gridVals;
  {{ 0 },
   { 0.785398 },
   { 1.5708 },
   { 2.35619 },
   { 3.14159 },
   { 3.92699 },
   { 4.71239 },
   { 5.49779}}
In[3]:
Example::resizeGrid(-1);
### Error 81: Illegal Number of Grid Points.
      Grid size < 0 was passed to Example::resizeGrid.
      File "Interactive"; Line 3 # caused the exception.
      File "ExampleCBase.cc"; Line 198 # generated the exception.
In[4]:
Example::resizeGrid(4);
In[5]:
Example::gridVals;
  {{ 0 },
   { 1.5708 },
   { 3.14159 },
   { 4.71239 }}
In[6]:
```

A more elaborate example that exercises the *Optimize* module is contained in *opt.sc*.

Under most circumstances, it *is* possible to omit the module name qualification in front of the member name. For instance,

```
In[1]:
ngpts();
  8
In[2]:
```

calls the desired function `Example::ngpts()`. The reason the qualification can sometimes be neglected is because of a *search list* maintained by the shell.

When an un-qualified name is typed, the shell first attempts to find this name in its database given the current scope level. If this search fails, the shell then sequentially goes through the search list looking for module qualifications that would result in a successful lookup. If, a *single* match is found, the shell assumes that the variable desired *implicitly* possessed the required qualification. In the example above, the shell found that no `ngpts` was defined at global scope and, moreover, that one, and only one module possessed a member by that name. Therefore, the function was called. If a version of `ngpts` existed at global scope, that version would have been called instead. If two modules had members named `ngpts` an error exception would have been generated. In this case, the module qualification must be used so that the shell knows which member is desired. An example of the issues involved is contained in the example:

```
In[1]:
Integer ngpts() { return 7; }
In[2]:
ngpts();
  7
In[3]:
Example::ngpts();
  8
In[4]:
```

By default *MGen* adds all modules to the search list. Therefore, one can usually omit the qualification unless trying to access a name defined simultaneously in two modules.

Note: you can also use the `list` and `about` functions described in Section 4.2.2 to find out the structure of Modules.

Chapter 4

Standard Modules

4.1 The Consts Module

One often needs to use mathematical and scientific constants such as π and μ_0 in one's code. According, we include a very simple, data-only, module giving various useful values. See the module description file "Consts.mod" to see the current set of available constants.

4.2 The Sys Module

The *Sys* module gives an interface to various system functions controlled by the shell. The functions fall into a number of categories:

- Array resizing.
- Database browsing.
- Date and time information.
- Error trapping.
- Timing.
- Version information.

Let us describe these functions in more detail.

4.2.1 Array Resizing

It is possible to resize a `RealArray3` object using the function

```
Void ra3Resize(const RealArray3 &a, const Integer &dim1,
  const Integer &dim2, const Integer &dim3)
```

where `dim1`, `dim2`, and `dim3` are the new dimensions. To resize `ComplexArray3` objects use `ca3Resize` and to resize `IntegerArray3` objects use `ia3Resize`.

To resize matrices use `cmResize`, `imResize`, and `rmResize`. These routines are similar to `ra3Resize` except, of course, they only expect two dimensions.

To resize vectors use `cvResize`, `ivResize`, and `rvResize`. These routines are similar to `ra3Resize` except, of course, they only expect one dimension.

All of these routines can be called from Fortran.

4.2.2 Database Browsing

It is possible to find out about classes and variables while running the SuperCode using the `list` abd `about` functions. The `list` function is used to find out the names of symbols currently defined in the SuperCode. This function has the prototype:

```
Void list(const String &);
```

The string should be a regular expression. (See any UNIX book for a discussion of what these are.) The `list` function outputs all symbols in the SuperCode database that match the specified regular expression. For example, try the following

```
list(".*"); // Lists all global symbol names.
list("a.*"); // Lists all global symbols starting with an 'a'.
list(".*::"); // Lists all type and module names.
list("Array3::d.*"); // Lists all members of the Array3 class
                     // starting with a 'd'.
list(".*::ctor"); // Lists all the 'ctor' functions in all classes.
```

A complementary function is `about`. This has the prototype

```
Void about(const String &);
```

The String in this case is a symbol name possibly found by `list`. For names that represent variables, the type and a description of the variable are printed. For function names, the return type, argument types, and a description are printed. For classes, the types and descriptions about all members is printed. For example, try typing

```
about("_i"); // Tells about the global variable '_i'.
about("exp"); // Tells about the function 'exp'.
about("Complex"); // Tells about the Complex class.
about("Vector::length"); // Tells about the member function.
about("ctor"); // Tells about all 'ctor' member functions.
```

Remember, the string passed to `about` is not a regular expression.

4.2.3 Date and Time Information

It is possible to get a string representation of the current date and time by calling the function

```
Void dateTime(String &d)
```

where `d` is a string to store the information.

4.2.4 Error Trapping

By default, the shell catches exceptions such as floating point errors, etc. Sometimes, for debugging purposes, it is best to allow the shell to dump core when an error occurs. This can be done by calling the function

```
Void errorTrap(const Integer &onoff)
```

with the boolean value `False`. To turn error trapping back on, call `errorTrap` with `True`.

4.2.5 Timing

One often wants to gather timing information. This can be accomplished by first calling the routine `timerOn()` and then, after the task that is being timed, the routine `elapsedTime()`, which returns a the time, in seconds, since the call to `timerOn`.

4.2.6 Version Information

It is possible to get a string representation of the shell version by calling the function

```
Void shellVersion(String &d)
```

where `d` is a string to store the information. The routine `applVersion` returns the application version information.

4.3 The Plot Module

The ability to make plots of data is an important capability for an interactive scientific code. Therefore, let us describe a plotting facility, implemented in the *Plot* module and shell code, supporting X–Y plots, contour plots, and 3–D surface plots. This module is based on the PLPLOT library, developed by LeBrun, *et al.* [9].

The goals for plotting facility's user interface are simplicity, expressiveness, flexibility, and compatibility. Our interface should encourage use by making it easy to make a simple plot. On the other hand, it should allow for the construction of complicated plots with a variety of different characteristics. Finally, we want the interface to be consistent with C++ syntax (i.e., no changes to the Shell language required).

4.3.1 Basic X–Y Plots

The simplest sort of plot involves graphing a series of ordinate (Y) values against a series of abscissae (X) values. We call this an X–Y plot. All X–Y plots are made using a call to the routine `xyPlot`. This routine has the prototype

```
Void xyPlot(const RealMatrix &y, const RealMatrix &x, option-list)
```

where

- `y` is an $N \times NX$ matrix holding NX ordinate arrays of length N.
- `x` is either a vector of length N or a $N \times NX$ matrix; `x` holds either one or NX abscissae arrays.

The *option-list* refers to an optional set of additional arguments that we will consider later.

The simplest X–Y plot involves plotting one vector versus another:

```
xyPlot(y, x);
```

creates a graph of $y(x)$. It is also possible to plot several curves at once. This is accomplished using the matrix concatenation operator:

```
xyPlot(y1 & y2, x);
```

displays $y_1(x)$ and $y_2(x)$ on the same plot. Alternatively, a matrix, whose columns hold the ordinate arrays, can be passed as the first argument.

For most applications, `x` is a vector. However, it is sometimes convenient to have a different abscissa vector associated with each ordinate. For example

```
xyPlot(y1 & y2, x1 & x2);
```

displays $y_1(x_1)$ and $y_2(x_2)$ on the same plot. In this case, `y` and `x` must have the same dimensions. An example of a case where this capability is useful is for plotting flux surface shapes. Here, we have a series of (R, Z) pairs describing each flux surface and many of these series describing the complex flux surface geometry.

4.3.2 Basic Contour Plots

Contour plots are made using a call to the routine `contourPlot`. This routine has the prototype

```
Void contourPlot(const RealMatrix &z, const RealVector &x,
  const RealVector &y, const RealVector &clevels, option-list
```

where

- `z` is an $M \times N$ matrix holding values of a function $z(x, y)$ on a grid.
- `x` is a vector of length M holding the x-values z is evaluated on.
- `y` is a vector of length N holding the y-values z is evaluated on.
- `clevels` is a vector of length NC holding the desired contour levels.

The *option-list* refers to an optional set of additional arguments that we will consider later. The values in the `x` and `y` arrays need not be evenly spaced. However, there is no mechanism for handling missing data in the `z` matrix.

4.3.3 Basic 3–D Plots

3–D plots are made using a call to the routine `threeDPlot`. This routine has the prototype

```
Void threeDPlot(const RealMatrix &z, const RealVector &x,
  const RealVector &y, const Real &altitude, const Real &azimuth, option-list
```

where

- `z` is an $M \times N$ matrix holding values of a function $z(x, y)$ on a grid.
- `x` is a vector of length M holding the x-values z is evaluated on.
- `y` is a vector of length N holding the y-values z is evaluated on.
- `altitude` is the altitude angle, in degrees, of the point the plot is being viewed from.
- `azimuth` is the azimuth angle, in degrees, of the point the plot is being viewed from.

The *option-list* refers to an optional set of additional arguments that we will consider later. The values in the `x` and `y` arrays need not be evenly spaced. However, there is no mechanism for handling missing data in the `z` matrix.

4.3.4 Options

In addition to simply plotting curves, one usually wants to add, for instance, labels, a title, colors, line styles, marks, etc. This is accomplished using the *option-list* referred to above. An *option-list* is a comma separated list of zero to ten *option-specifications*. An *option-specification*, in turn, has the syntax

option = *expression*

where an *option* is simply an identifier. Therefore, an example of an X–Y plot call with a non-empty *option-list* is

```
xyPlot(y, x, xLabel = ``x'', yLabel = ``y'', lineColor = red);
```

This gives x-axis and y-axis labels and indicates that the curve should be drawn in red.

The possible options are summarized below.

- `axesColor` (XY, CP, 3D[1]) Sets the color of the axes. Possible choices are: `coral`, `red`, `yellow`, `green`, `aquamarine`, `pink`, `wheat`, `grey`, `brown`, `blue`, `blueViolet`, `cyan`, `turquoise`, `magenta`, `salmon`, and `white`. Default value is `white`.

- `scale` (XY, CP) Sets scaling of the axes. If `scale` is `normal`, the axes are scaled independently to use as much of the screen as possible; if `scale` is `equal` the scales of the x and y axes is made equal. Default value is `normal`.

- `axes` (XY, CP) Controls drawing of the box (axes) around the plot. Possible choices are `noBox` (no axes or annotation), `boxOnly` (draw the bounding box only), `normalAxes` (indicating draw the bounding box with coordinate values around the edge), `axisX0Y0` (same as `normalAxis` but also draw the lines $x = 0$ and $y = 0$), `axisX0Y0Grid` (same as `axisX0Y0` but also draw a grid), `logLin` (logarithmic x axis, linear y axis), `logLinY0` (same as `logLin` but also the line $y = 0$), `linLog` (linear x axis, logarithmic y axis), `linLogX0` (same as `linLog` but also the line $x = 0$), and `logLog` (logarithmic x and y axes). Default value is `normalAxes`.

- `labelColor` (XY, CP, 3D) Sets the color of the axis labels and the title. See the description of `axesColor` for possible choices. Default value is `white`.

- `labelFont` (XY, CP, 3D) Sets the font of the axis labels and the title. Possible choices are `simple` (a sans serif font), `italic`, `roman`, and `script` (a handwriting font). Default value is `simple`.

- `xLabel` (XY, CP, 3D) Sets the text for the x axis label. Any character string can be assigned. It is possible to include escape sequences to trigger greek letters, subscripts, superscripts, etc. This will be discussed later. Default value is the empty string.

- `yLabel` (XY, CP, 3D) Sets the text for the y axis label. Any character string can be assigned. Default value is the empty string.

[1]XY = applicable to X-Y plots, CP = applicable to contour plots, 3D = applicable to 3–D plots.

- `zLabel` (3D) Sets the text for the z axis label. Any character string can be assigned. Default value is the empty string.
- `title` (XY, CP, 3D) Sets the text for the plot title. Any character string can be assigned. Default value is the empty string.
- `mark` (XY) Sets the symbol used to mark data points. Possible choices are `noMark`, `box`, `plus`, `asterisk`, `dot`, `circle`, `cross`, `triangle`, `crossHair`, `bullsEye`, `diamond`, `star`, `solidBox`, `solidCircle`, `solidStar`, or `cycle`. The `cycle` option is special: it will be described in more detail later. Default value is `noMark`.
- `markColor` (XY) Sets the color of the marks. See the description of `axesColor` most possible choices. In addition, the user may set `mark` to `cycle`. Default value is `white`.
- `lineColor` (XY, CP, 3D) Sets the color of the lines. See the description of `axesColor` most possible choices. In addition, the user may set `lineColor` to `cycle` for X–Y plots. Default value is `white`.
- `lineStyle` (XY, CP, 3D) Sets the style of the lines. Possible choices are `solid`, `dashedN` (where `N` is `1`–`7`), or `cycle`. The `cycle` value applies to X–Y plots only. Default value is `solid`.
- `surface` (3D) Sets cross-hatching on 3–D surface. Possible choices include `xLines` (lines are drawn parallel to x axis), `yLines` (lines are drawn parallel to y axis), and `xyLines` (lines are drawn parallel to x axis and y axis). Default value is `xyLines`.
- `meshPlot` (3D) Tells whether or not to plot a 3–D surface mesh. Possible choices are `yes` and `no`. Default value is `no`.

One can include escape sequences in the character strings that are assigned to the `xLabel`, `yLabel`, `zLabel`, and `title` options. All escape sequences start with the number symbol (#). The following escape sequences are possible:

- `#u` Move up to the superscript position (ended by `#d`).
- `#d` Move down to the subscript position (ended by `#u`).
- `#b` Backspace (to allow overprinting).

- `##` The number symbol.
- `#+` Toggle overline mode.
- `#-` Toggle underline mode.
- `#gx` Greek letter corresponding to the roman letter `x` (experiment to figure out the mapping).
- `#fn` Switch to simple font.
- `#fr` Switch to roman font.
- `#fi` Switch to italic font.
- `#fs` Switch to script font.

For example, "`#gk#da#u`" will lead to κ_a

The options `mark`, `lineColor`, `markColor`, and `lineStyle` can be assigned the special value `cycle`. This value applies only to X–Y plots and is useful in when multiple curves are being plotted. In the case of the `mark` option, this value indicates that the mark should cycle through the series of values stored in the array `Plot::markCycle`. The first curve will be marked according to `Plot::markCycle(1)` and so on until the end of the array is reached. Then, the marks will restart the cycle at the beginning. The same goes for `lineColor` and `markColor` (which cycle through the `Plot::colorCycle` arra) and `lineStyle` (which cycles through the `Plot::styleCycle` array. The user can adjust how the cycle goes by resizing and adjsting the contents of the clycle arrays.

Examples of plotting routine usage are found in the Shell file "plot.sc."

Bibliography

[1] Bjarne Stroustrup: *The C++ Programming Language*, Addison-Wesley, Reading, Massachusetts, 1986.

[2] Brian W. Kernigan and Dennis M. Richie: *The C Programming Language*, Prentice-Hall, Englewood Cliffs, New Jersey, 1983.

[3] Samuel P. Harbison and Guy L. Steele: *C, A Reference Manual*, Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[4] Bjarne Stroustrup: *The C++ Programming Language*, Addison-Wesley, Reading, Massachusetts, 2nd Edition, 1991.

[5] Stanley B. Lippman: *The C++ Primer*, Addison-Wesley, Reading, Massachusetts, 2nd Edition, 1991.

[6] Margaret A. Ellis and Bjarne Stroustrup: *The Annotated C++ Reference Manual*, Addison-Wesley, Reading, Massachusetts, 1990.

[7] Grady Booch: *Object Oriented Design with Applications*, Benjamin/Cummings, Redwood City, California, 1991.

[8] Paul F. Dubois, Lee Busby, Peter Willmann, and Janet Takemoto: *MPPL, A More Productive Programming Language.* Available electronically from any NERSC computer by typing first "`cfs get /basis/mppldoc/mpplrpt`" and then "`allout nip mpplrpt gr. box` *ann* `mppl`" where *ann* is your box number.

[9] M.J. LeBrun, G. Furnish, and T. Richardson: *The PLPLOT Plotting Library Programmers Reference Manual Version 4.0*, University of Texas at Austin Report DOE/ET-53088-538, 1992.