Appendix A: A Brief Summary of the Lua Programming Language

The programming language Lua is a relatively recent addition to the repertoire of computer programming languages. The first version (1.1) was released in July 1994. The most recent version of the language is version 5.1 (as of early 2008). Since its introduction it has assumed a dominant role as a favorite scripting language for computer games, to a large extent because of its small code size and fast execution speed. The major reasons for selecting this language for implementing the computer algorithms of this work are summarized in Chapter 2. The discussion here provides a more in depth summary of the most important features of the Lua language. However, the following two very excellent, in depth discussions are readily available on the web:

1. Lua Reference Manual -- http://www.lua.org/manual
2. Programming in Lua -- http://www.lua.org/pil

The material in this Appendix is a somewhat shortened version of the Lua Reference Manual and the reader is referred to this material for a much more complete discussion of the Lua language

Lua has been described by its developers as "an extension programming language designed to support general procedural programming with data description facilities". Many of the applications involve using Lua as a "scripting language" embedded within another language such as the C programming language. In fact Lua is written and compiled in clean C which is a common subset of ANSI C. Lua is a freely distributed programming language and a copy of the software is supplied on the disk accompanying this book. The user is advised to consult the Lua web site for possible later additions to the software (www.lua.org).

A.1 Lua Language Fundamentals

Names in Lua can be any string of letters, digits and underscores, not beginning with a digit. The exception to this rule is the set of following *keywords* that can not be used as object names:

```
and       break     do        else      elseif
end       false     for       function  if
in        local     nil       not       or
repeat    return    then      true      until
while
```

Lua is case sensitive so that although `end` is a reserved word, `End` and `END` are perfectly valid names. As a convention, names beginning with an underscore followed by all capital letters are reserved for internal Lua usage and should not be used as naming conflicts may occur. Most of the reserved words are similar to reserved words in other computer languages and the list should be familiar to anyone familiar with another programming language.

In addition to the keywords, the following tokens also have special meaning in Lua:

```
+     -      *      /      %      ^      #
==    ~=     <=     >=     <      >      =
(     )      {      }      [      ]
;     :      ,      .      ..     ...
```

Again most of these symbols have the same meaning as in other languages. One exception is perhaps the use of ~= for "not equal".

Literal strings are delimited by matching single or double quotes and can contain C-like escape sequences such as '\b' (backspace), '\n' (newline) and '\t' (tab). A character in a string may also be specified by its numerical value using the escape sequence \ddd where ddd is a sequence of three decimal digits representing the character. Literal strings may also be defined using a long format of double brackets or double brackets with an intervening == string as for example [[text string]] or [==[text string]==].

Numerical constants consist of a sequence of numerical digits with an optional decimal part and an optional decimal exponent. Integer hexadecimal constants are also accepted if the digits are prefixed with 0x. Some valid numerical constants are:

```
5   5.00  0.005 3.33e-4     3.33E-4     0xff  0x4a
```

Internally Lua represents integers as simply double precision numerical numbers. Limits to the representation of numerical constants are discussed and explored in Chapter 2.

Lua comments begin with a double hyphen (--) anywhere outside a string. If the text following the double hyphen is not an opening long bracket, the comment is known as a *short comment* and extends until the end of the code line. Otherwise it is a *long comment* (beginning with --[) that runs until the corresponding closing long bracket (--]). Such long comments are very useful for extended comments in code or to temporarily disable sections of code extending over many lines of text.

Lua is a *dynamically typed language* and as such variables do not have types, only values have types. Because of this there are *no type definitions* in Lua and values carry their own type. For many users accompanied to other languages this takes some getting used to. However, it is one of the great features of the language as all values are *first-class values*. All values in the language may be stored in variable names, passed as arguments to other functions or returned as results.

Lua has eight basic types of values: *nil, Boolean, number, string, function, userdata, thread and table.* The type *nil* is that of the reserved work *nil* and usually represents the absence of a useful value, for example the value of an unassigned variable. A *Boolean* type has only the two values *false* and *true*. The types

number and string should be familiar to anyone that has used a programming language. A variable of type *function* can be a code segment written in Lua or a function written in the C language. Functions are called with an argument list and functions may return any number of variables. The types *userdata* and *thread* are important types for embedded applications but are not used in the programs written in this work and thus are not discussed further here. The type of a value can be obtained by the function type(val) which returns a string identifying the type.

The fundamental data structure in Lua is the *table* and in fact this is the *only data structure in Lua*. While this is somewhat disconcerting to some new user of Lua it is one of the most useful features of Lua. The type table implements *associative arrays* which means that tables can be indexed not only with numbers but with any other value (except nil). Tables in Lua may be used to implement ordinary arrays, symbol tables, sets, records, graphs, trees, vectors, matrices, etc. In Lua tables are simply objects and the name of a table does not store the table – it simply contains a reference or pointer to the table. Tables can thus be readily passed to functions with a minimum of expense in computing resources.

Table constructors are expressions that create tables and are specially formatted expressions enclosed within braces ({ }). An example to create a table named tbl is:

```
tbl={3, 5.66, ['one'] = 1.0, ['name'] = 'John', [30] = 33}
```

This creates a table with three integer fields (or indices) (1, 2 and 30) and with two string fields ‘one’ and ‘name’. Examples of accessing the field values are: tbl[2] (value 5.66), tbl[‘one’] (value 1.0) and tbl[‘name’] (value ‘John’). For table fields with string identifiers, Lua provides another more convenient mechanism of obtaining the value as for example tbl.one (value 1.0) or tbl.name = ‘Joe’ (changes string ‘John’ to string ‘Joe’). All global variables in a Lua program live as fields in Lua tables called environment tables. A table field may reference another table and tables may be embedded within tables to any desired depth.

Variables in Lua are names of places that store values. They are of three different flavors: global variables, local variables and table fields. By convention, variables are global in scope unless limited in scope by the *local* keyword. Note that there is no *global* keyword as this is the default case.

A.2 Lua Statements

Statements in Lua follow almost the same form as in C and as other conventional programming languages. The basic unit of execution in Lua is called a *chunk* which is simply a sequence of Lua statements. Each statement in Lua may be optionally followed by a semicolon, although the normal convention in Lua programming is to only use the semicolon if an additional statement is placed on the same programming line. No empty statements are allowed, so that ; ; is a programming error.

Closely related to a chunk is the concept of a *block* which syntactically is the same as a chunk. Again a block is a list of Lua statements. A block may be ex-

plicitly delimited to produce a single statement of the form **do** block **end** where the **do** ... **end** structure represents a single statement. Such explicit blocks are useful to control the scope of variables as variables may be declared as *local* to the block of code.

Perhaps the most fundamental Lua statement is the assignment statement which contains a list of variables on the left side of an equal sign and a list of expressions on the right side of the equal sign. The assignment statement first evaluates all the expressions and only then performs the assignments. Some examples are:

```
i = 4
x, y, z = y, x, i
i, a[i] = i + 1, 10
```

In the second line the values stored in x and y are interchanged and z is set to the value of i. In the third line i is set to 5 and a[4] is set to 10. In an assignment statement the number of variables on the left may differ in number from the number of expressions on the right. If the number of variables is less than the number of expressions, the extra expressions are not evaluated. If the number of variables exceeds the number of expressions, the extra variables are set to **nil** values. The meaning of an assignment statement to a table field may be changed by use of a metatable as subsequently discussed.

Lua has several flow control structures that have similar meanings to those in other languages. The most important are:

while *exp* **do** *block* **end**
repeat *block* **until** *exp*
if *exp* **then** *block* [**elseif** *exp* **then** *block*][**else** *block*] **end**

For the **if** statement, the expressions in brackets may or may not be present and any number of **elseif** clauses may be present. One of the differences with some other languages is the use of the **end** keyword to terminate the control structures. For the test expressions (*exp* terms in the above) any value may be evaluated by the expression with both **false** and **nil** considered as false. Any other evaluated value is considered as **true**. In the **repeat** structure the terminating *exp* can refer to local variables declared within the repeating block.

In addition Lua has two types of **for** control statements. One is a numeric for statement of the form:

for var_name = *exp1*, *exp2* [,*exp3*] **do** *block* **end**

The block of code is repeated for var_name equal to *exp1* to *exp2* with the variable stepped by value *exp3* between each repeat execution. The default value of *exp3*, if omitted is 1. The control expressions are evaluated only once before the block is executed and they must all result in numbers. The loop variable var_name is considered local to the **for** loop and the value is not retained when the loop exits.

The generic **for** statement executes over functions called *iterators* and has the form:

for *namelist* **in** *explist* **do** *block* **end**

This type of **for** statement is not available in many common programming languages and takes some understanding for those first encountering Lua. An expression such as:

for *var_1, ..., var_n* **in** *explist* **do** *block* **end**

is best understood as equivalent to the following code segment:

```
do
   local f, s, var = explist
   while true do
      local var_1, ..., var_n = f(s, var)
      var = var_1
      if var == nil then break end
      block
   end
end
```

As the expanded equivalent indicates the evaluation of *explist* must return an *iterator* function (the *f* term), a *state variable* (the *s* term) and an *initial value* (the *var* term) for the *iterator variable*. One of the common uses of the generic **for** loop is to iterate over the values of a table and perform some operation on the table values. The reader is referred to the Lua programming manual for a more complete discussion of this control structure.

The keyword **break** may be used to exit control loops but must be the last statement in a block. Because of this restriction the construct **do break end** must sometimes be used to break from within the middle of a block of code.

A.3 Lua Expressions

Expressions constitute the fundamental construction unit for code in Lua. Such expressions have been indicated in the control structures of the preceding section. In its simplest form an expression consists of Lua keywords such as **true**, **false** or **nil**. Expressions may also be numbers, strings, functions, table constructors or variable names representing such values. Expressions may also be more complex units of code such as two expressions between a binary operator, such as *exp =exp1 + exp2*. In general Lua expressions are similar to the fundamental expressions of the other languages.

The Lua evaluation of an expression typically results in one or more Lua object types being returned by the evaluation. The resulting evaluated items are then used by Lua as control parameters as in the previous section or as values to be set to other variables by use of the = construct as in *exp = exp1 / exp2*.

In Lua both function calls (see next section) and expressions may result in multiple values being evaluated. Lua has a specific set of rules as to how such multiple values are subsequently used by Lua code. This is necessitated by the fact that

Lua also allows multiple assignment statements. Some Lua statements and the results are shown below:

```
a, b, c = x, y, z    -- a, b and c set to value of x, y and z
a, b, c = x          -- a set to value of x, b and c set to nil
a, b = x, y, z       -- a and b set to value of x and y, value of z is
                        discarded
```

The more complex rules involve multiple values returned by function calls (such as f()). If a function call is used as the last (or only) element of a list of expressions then no adjustment is made to the number of returned values. In all other cases Lua adjusts the number of returned elements by a function call to one element, discarding all returned values except the first returned value. Some examples and the resulting adjustments are:

```
f()              -- adjusted to 0 results when used as an isolated statement
a, b, c = f()    -- f() is adjusted to 3 values adding nil values if needed
a, b, c = f(), x -- f() is adjusted to 1 value, b set to x value and c gets nil
a, b, c = x, f() -- f() is adjusted to 2 values, a is set to x value
g(x, f())        -- g() gets x plus all returned values of f()
g(f(), x)        -- f() is adjusted to 1 value and g() gets two parameters
return f()       -- returns all results of f()
return x, y, f() -- returns x, y, and all results of f()
return f(), x, y -- f() is adjusted to 1 value and three values are returned
{a, b, f()}      -- creates a list with elements a, b and all returned values of f()
{f(), a, b}      -- f() is adjusted to 1 value and a list of three elements is created
{f(), nil}       -- creates a list with 1 element, the first value returned by f()
```

An expression enclosed in parentheses always returns one value so that (f(x, y, z)) results in only one value regardless of the number of values returned by f().

Lua supports the usual array of arithmetic operators: + for addition, - for subtraction, * for multiplication, / for division, % for modulo, ^ for exponentiation and unitary – before any number. The supported relational operators are:

```
==    ~=    <    >    <=    >=
```

with the resulting evaluation resulting in a Lua **false** or **true** value. For equality the type of the operands must be the same. Objects such as tables and functions are compared by reference and two objects are considered equal only if they are the identically same object. For example two different tables with the same identical table values will not be considered as equal. The way that Lua compares tables can be changed by the use of metatables. The operator ~= is the negation of equality (the == operator).

The logical operators in Lua are the keywords **and, or** and **not**. These consider both **false** and **nil** as false and everything else as true. The **and** operator returns its first argument if it is **false** or **nil** and otherwise returns its second argument. The **or** operator returns its first argument if its value is different from **false** and **nil** and otherwise returns its second argument. Statements such as "a = a or 1" are frequently used in Lua to ensure that a variable (such as a) has either a defined value or is set to a default value of 1 if not previously defined.

For string concatenation Lua uses the '..' operator. If one of the operands is a number that can be converted to a string, Lua will perform the string conversion before applying the operator.

Lua has a special length operator, #, for obtaining the length of an object. The length of a string is the number of bytes. The length of a table t is defined as any integer index n such that t[n] is not **nil** and t[n+1] is **nil**. For a table with no **nil** values from 1 to n this gives the expected length of the table n. The operator causes some confusion when applied to tables with missing or **nil** values and for tables indexed by strings such as dictionary tables. Lua provides other mechanisms such as pairs() and ipairs() for stepping over such tables and the length operator should only be applied to tables indexed by numerical values with no missing elements.

Lua observes the following operator precedence with the order from lower to higher priority:

or
and
< > <= >= ~= ==
..
\+ -
* / %
not # - (unary)
^

Parentheses can be used to change the precedence of an expression.

A.4 Function Definitions and Function Calls

Functions are fundamental code blocks in any programming language leading to the ability to produce modular code segments that can be used in a variety of application. Two acceptable formats for function definitions are:

fct = **function** (parlist) block **end**
function fct (parlist) block **end**

where fct is a user supplied name for a function, parlist is a comma separated list of calling arguments and block is the Lua code to be executed by the function. When Lua encounters the function statement, the code is simply compiled by Lua and a later call to the function is used to explicitly execute the function. Function names in Lua are first class objects and can be stored in tables and passed to other functions just as any other variable in Lua. The keyword local can also be used to precede the function definition to limit the scope of the function name.

When Lua encounters a function definition and compiles the function it establishes an environment for the function that consists of the state of any global variables used within the function body. This means that different instances of the same function may refer to different external variables and may have different en-

vironmental tables. This is an important feature of Lua not available to statically compiled languages.

In defining a function the argument list (parlist above) is a list of comma separated Lua variable names. These names are treated as local variables to the function body and are initially set to the argument values passed to the function by a call to the function. The argument definition may (if desired) contain as the last argument in the list an entry consisting of three dots, i.e. '...'. This indicates that the function is a *vararg function* and can be passed a varying number of calling arguments. It is up to the called program to determine the number of such variable arguments.

In calling for the execution of a previously defined function the argument list used in calling the function is a list of comma separated statements that must evaluate to one of the eight basic Lua types of objects. For a function with a fixed number of calling arguments, the list of calling arguments is adjusted to match the number of calling arguments used in the defining statement. This is achieved by dropping arguments if the number exceeds the number in the defining code or adding nil arguments if the number is less than that in the defining code for the function. For a vararg function, all extra arguments are collected into a *vararg expression* that is also written as three dots. In this manner a function can accept a varying number of arguments in a calling statement. Lua makes no check on the type of argument used in calling a function to verify that they match the type of arguments used in defining the function. This is all left up to the user.

Object oriented programming in Lua typically involves the use of a table to store several functions that are associated with an object as well as storage of the object parameters with the same or another table. Recall that tables are the only data object defined in Lua. Lua provides a convenient means for references such named functions within tables. For example consider a table defined as C = {f1, f2, f3} where f1, f2 and f3 are names of functions that have previously been defined. Calls to the functions can then be made using the statements C.f1(), C.f2() or C.f3() with appropriate argument lists. The notation C.f1 is simply shorthand notation for C['f1'] which returns a reference to the f1 function. In defining methods for objects, Lua provides the syntactic sugar for the statement v:name(args) as equivalent to v.name(v, args). This is typically used in combination with metamethods for tables as described in the next section.

Values are returned by a function call with the statement **return** explist where explist is a comma separated list of Lua objects. As with break, the return statement must be the last statement of a block of code. In some cases this necessitates the **do return** explist **end** Lua construct to return from within a block. Lua implements function recursion and when the recursive function call is of the form **return** function_call() Lua implements proper tail calls where there is no limit on the number of nested tail calls that a program can execute.

Calls to evaluate a function may be made as simple single line statements in which case all values returned by the function are discarded. When a list of arguments is set equal to a function call, the number of returned arguments is adjusted to match the number of arguments in the left hand list of values.

A.5 Metatables

Every value in Lua may have an associated metatable that defines the behavior of the value under certain operations. Such metatables are typically used to define how table objects behave when used with certain inherent Lua operations on the table. For example if a non-numeric value is the operand in an addition statement, Lua checks for a metatable of the operand value with a table field named "__add". If such a field exists then Lua uses this function (__add()) to perform the addition function. The use of metatables is one of the features that makes Lua such an extensible language as tables can be used to define any type of object and metatables may be defined to indicate how basic language operations behave for the defined objects.

The keys in a metatable are called *events* and the values (typically functions) are called *metamethods*. For the example above the event would be the "add" operation and the metamethod would be the __add() function defined to handle the event. Values in a metatable can be set or changed by use of the Lua setmetatable(val, mtable) function and values may be queried through the getmetatable(val) function that returns the metatable itself.

Lua supports several standard metamethods for metatables. These are identified to Lua by a string name consisting of two initial underscores followed by a string identifier, such as "__add". The string must subsequently be the name of a defined function that handles the indicated operation for the Lua object with the given metatable. Lua supports the following metamethods:

"—add" : the addition or + operation
"__sub" : the subtraction or – operation
"__mul" : the multiplication or * operation
"__div" : the division or / operation
"__mod : the modulo or % operation
"__pow" : the power or ^ operation
"__unm" : the unary or – operation
"__concat" : the concatenation or .. operation
"__len" : the length or # operation
"__eq" : the equal or == operation
"__lt" : the less than or < operation
"__le" : the less than or equal or <= operation
"__index" : the table access or indexing operation (table[key] operation)
"__newindex" : the table assignment operation (table[key] = value operation)
"__call" ; the table call operation (used as table())

By defining a set of such functions for table objects, Lua provides an extension language for the operations indicated above to be automatically performed between newly defined language objects. This combined with the storing of func-

tions as names within tables provides a powerful mechanism for object oriented programming in Lua.

A.6 Environments

Lua objects of type thread, function and userdata have another table associated with them called their *environment.* For the purpose here only function environments will be discussed. The environment table for a function defines all the global variables used to access values within the function. The environment of a function can be set with the statement "setfenv(fct, table)" and the environment table can be accessed by the statement "getfenv(fct)" which returns the environment table. An explicitly defined environmental table for a function is an excellent means of ensuring that the function does not define and export any undesirable global variables to a Lua program that uses the function.

A.7 Other Lua Language Features

Lua performs automatic garbage collection so the user does not have to worry with allocating and de-allocating memory as in many languages (such as C). None of the programs in this book worry with garbage collection but there are Lua functions for forcing garbage collection if needed. The reader is referred to the Lua manual for more details.

Lua supports coroutines that provide a form of *collaborative multithreading* of Lua code. By use of functions coroutine.yield() and coroutine.resume() Lua can be forced to suspend execution from one function and later resume the execution from within another Lua function. For some applications such an ability is very useful but this feature is not used in this work.

Lua has an extensive set of C Application Program Interface (API) functions for interfacing Lua with C language programs. C functions may be called from Lua programs or Lua programs may be called from C programs. The usefulness of this API is one of the Lua features that has made it the dominant scripting language for computer games. However, this feature is not used here so the reader is again referred to the Lua manual and Programming in Lua book for a discussion.

A.8 Lua Standard Libraries

In addition to the basic Lua language features, Lua provides several sets of standard libraries. Every Lua library is defined by an associative Lua table with the names of the functions as an index string into the table and the function is the value stored at the given index. The standard Lua libraries are grouped as follows:

- basic library (_G[])
- package library (package[])
- string manipulation (string[])

- table manipulation (table[])
- mathematical functions (math[])
- input and output functions (io[])
- operating system facilities (os[])
- debug facilities (debug[])

In the above the [] notation is used to indicate that the name[] is a table of values. These functions are implemented with the official C API and are provided as separate C modules. The standalone Lua execution program integrates all of these libraries into the executable program. The usage of Lua in this book assumes that these libraries are part of the Lua executable program although the debug facilities are not employed in this work.

The basic library functions are defined in a default Lua environment with the name _G. The table entries in the default environment may be observed from Lua by printing the entries in this table for example by execution the statement "for name, value in pairs(_G) do print(name) end". The following table entries are found in the _G table corresponding to the variable names in the standard Lua environment:

basic Lua environment names:

_G[], _VERSION<$, arg[], assert(), collectgarbage(), coroutine[], debug[], dofile(), error(), gcinfo(), getfenv(), getmetatable(), io[], ipairs(), load(), loadfile(), loadstring(), math[], module(), newproxy(), next(), os[], package[], pairs(), pcall(), print(), rawequal(), rawget(), rawset(), require(), select(), setfenv(), setmetatable(), string[], table[], tonumber(), tostring(), type(), unload(), unpack(), xpcall()

The names shown are not exactly the string names stored in the _G[] table but have been augmented at the end with a 2 character code to indicate which of the 8 fundamental Lua types of variable is stored in the table. For example the last two characters of [] indicate that the entry name refers to a table. The character codes and associated data types are: string = '<$', number = '<#', function = '()', table = '[]', userdata = '<@', thread = '<>', boolean = '<&'. The entries in the _G[] table are thus seen to be of type string, table or function. The use of most of the functions can be inferred from the names of the functions.

The functions and tables in the _G[] environmental table can be referenced in Lua by simply stating the name of the table entries. For example the version number can be printed by the statement "print(_VERSION)" and the type of a variable such as next() can be printed by the statement "print(type(next))".

The standard Lua libraries listed above are seen to be the names of entries in the _G[] table such as math for the math library. The functions supplied by any of the standard libraries can be observed by printing the corresponding table entries such as the statement "for name, value in pairs(math) do print(name) end" to observe the math table entries. The following is observed for the standard Lua library tables and corresponding functions:

- package library (package[])

  config<$, cpath<$, loaded[], loaders[], loadlib(), path<$, preload[], seeall()

- string manipulation library (string[])
  byte(), char(), dump(), find(), format(), gfind(), gmatch(), gsub(), len(), limit(), lower(), match(), rep(), reverse(), split(), sub(), trim(), upper()
- table manipulation library (table[])
  concat(), copy(), foreach(), foreachi(), insert(), inverse(), maxn(), remove(), reverse(), sort()}
- mathematical functions library (math[])
  abs(), acos(), asin(), atan(), atan2(), ceil(), cos(), cosh(), deg(), exp(), floor(), fmod(), frexp(), huge<#, ldexp(), log(), log10(), max(), min(), mod(), modf(), pi<#, pow(), rad(), random(), randomseed(), sin(), sinh(), sqrt(), tan(), tanh()
- input and output functions library (io[])
  close(), flush(), input(), lines(), open(), output(), popen(), read(), stderr<@, stdin<@, stdout<@, tmpfile(), type(), write()
- operating system facilities library (os[])
  clock(), date(), difftime(), execute(), exit(), getenv(), remove(), rename(), setlocale(), time(), tmpname()
- debug facilities library (debug[])
  debug(), getfenv(), gethook(), getinfo(), getlocal(), getmetatable(), getregistry(), getupvalue(), setfenv(), sethook(), setlocal(), setmetatable(), setupvalue(), traceback()

The majority of the library functions used in this work are reasonably self evident from the names of the functions or are explained in the text when encountered. Thus explanations of the various standard library functions will not be given here. The reader is referred to the Lua reference manuals for more details.

The conventional notation for referring to an entry in the _G[] table, such as the type() function would be the code _G.type(). This will certainly invoke the type() function. However, for simplicity Lua accepts references to entries in the _G[] table without the _G prefix so that the code statement type() can be used to invoke the function. For the other tables in the _G[] table, the standard Lua access methods must be used. For example to invoke the sin() function within the math[] table, the proper Lua code statement is “math.sin()”. Another example is “os.clock()” to get information about the time for use in timing the execution of a block of code. To simplify notation one can define additional names such as sin = math.sin so that the sin function can be accessed by the statement: sin().

Appendix B: Software Installation

This Appendix describes the software supplied with this book and describes the typical installation and use of the software programs. This file is also supplied on the CD as the readme.txt file.

B.1 Introduction to Software on CD

The files supplied on the accompanying CD are arranged in directories as follows:

```
readme.txt
setup.exe
Examples/
        Chapter1/
        Chapter2/
        Chapter3/
        Chapter4/
        Chapter5/
        Chapter6/
        Chapter7/
        Chapter8/
        Chapter9/
        Chapter10/
        Chapter11/
        Chapter12/
        Chapter13/
Nonlinear Models/
        EasyMesh/
        gnuplot/
        Lua-5.1.3/
        SciTE174/
```

The readme.txt file is a text version of this Appendix.

The computer programs needed to execute the computer code for the various chapter of this book can be downloaded to the reader's computer by executing the setup.exe file on the CD. During the installation the user has the option of selecting a "compact" installation that installs only the Nonlinear Models directory or a "full" installation that in addition installs the files in the Examples directory. Unless the user's computer is low on disk space it is highly recommended that the Examples files also be copied as these contain the example computer code dis-

cussed in each chapter of the book. After the installation, the user's computer should be configured for easy access to these files.

The computer programs associated with each chapter of the book are organized into chapter directories under the Examples directory. The user can either access these from the CD or download these to the user's computer if desired using the "full" installation.

The user may also manually copy the appropriate files to a chosen directory on his/her computer. In this case the user will need to manually configure some of the files to properly access the features of the chapter programs. The user might wish to do this manual installation if the Lua language is already installed on the computer. This manual procedure is described in a later section of this file.

The files under the Nonlinear Models directory are programs and files required to execute the Lua programs as supplied with this book and as implemented in the text. The files are organized into 4 major directories that are now briefly discussed.

The EasyMesh directory contains all the files associated with the EasyMesh software. This software is used in Chapter 13 to generate triangular spatial grids for use in solving partial differential equations in two spatial dimensions. Programs in this directory are a copy of software that is freely available on the web at: http://www-dinma.univ.trieste.it/nirftc/research/easymesh/Default.htm. The files are supplied here as a convenience to the user of this book.

The gnuplot directory contains all the files associated with the gnuplot software which is public domain software for plotting graphs from data files. This software is used throughout this book to provide pop-up graphs of the curves generated by many of the computer programs. This software is freely available on the web at: http://www.gnuplot.info. The files are supplied here as a convenience to the user of this book.

The SciTE174 directory contains all the files associated with the SciTE software which is public domain software for a programming oriented text editor. This is the highly recommended text editor for use with the Lua programs developed and used in this book. Although any text editor may be used for editing Lua programs, the SciTE editor provides language highlighting and an easy interface for program development and execution. The software is freely available on the web at: http://scintilla.sourceforge.net/SciTE.html. The files are provided here as a convenience to the user of this book.

The Lua-5.1.3 directory contains all the files associated with Lua which is public domain software for the Lua language used as the basis for all the software examples in this book. Reasons for selecting this language are discussed in Chapter 2 of the text. The software is freely available on the web at: http://www.lua.org. The files are provided here as a convenience to the user of this book.

The Lua-5.1.3 directory is arranged into subdirectories as follows:

```
Lua-5.1.3/
        doc/
        etc/
        src/
                lua/
                stdlib/
        test/
```

The doc directory contains files with more detailed descriptions of the Lua language and the user can read the documentation for a more complete description of the Lua language. Even more complete descriptions of the language are available on the web at:

Lua Refernce Manual -- http://www.lua.org/manual and
Programming in Lua -- http://www.lua.org/pil

The src directory contains the computer code for the Lua language which is written in the C language. From the files in this directory an executable Lua program can be compiled. However, the user of this software does not have to perform this step as a lua.exe file is supplied in this directory. This lua.exe file is the program used to execute all the Lua programs developed in this book.

The lua directory under the src directory is an important directory for using the Lua programs in the various chapters of this book. This directory contains various code segments supplied with this book and that are required for executing the Lua programs in the various chapters of this book. All of the files supplied in the lua directory are programs developed specifically for use with this book and are not files supplied with the Lua language. They are collected within the src/lua directory because this is the standard directory searched by the lua.exe program for such files.

B.2 Full Installation of Software

For this discussion it is assumed that most users of this book and of the supplied software will not have the Lua language previously loaded on their computer. Thus this first description of installing the software is for users who do not have the Lua language on their computer and need the full installation of the software. For those who already have the Lua language installed, please skip to the next section on a partial installation of the software.

Executing the setup.exe program on the CD should provide the needed installation of the software required for execution of all the Lua programs associated with this book. The user will be asked during the installation to select a directory for the software. A suggested default directory is C:\Program Files. If this default is used the software programs will be copied into the C:\Program Files\Nonlinear Models directory with the configuration discussed in the previous section. Entries

will also be made in the Registry such that the user can double click on a *.lua file and have the file automatically be loaded into the SciTE editor. The user can of course select a different major directory if desired, such as D:\Work in which case the software would be copied into the D:\Work\Nonlinear Models directory. The "full" software installation requires about 120MB of disk space on the selected drive while the "compact" installation requires about 35MB of disk space.

After installing the software with setup.exe on the disk, the user may verify the proper installation by the following procedure. Open MS Explorer (or another program) in the directory selected to download the disk files. Figure B.1 shows such a directory listing where the root directory is the default download directory of C:\Program Files\Nonlinear Models. The Folders listing on the left of the figure show the directories that should be created by the downloading process. The listing on the right shows the example code associated with Chapter 3 of the book with an arrow pointing to the list3_7.lua file.

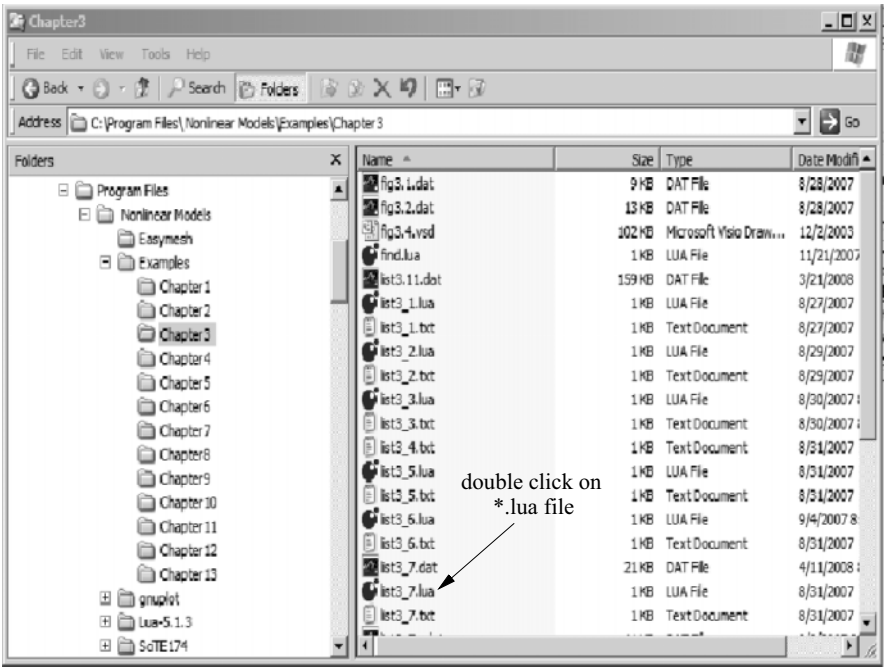

Figure B.1. Explorer listing of downloaded files. Directory is for C:\Program Files\Nonlinear Models\Examples\Chapter3.

If the software has downloaded properly and the computer is properly configured, a double mouse click (right button) on this *.lua file should bring up the SciTE editor with the Lua code from the file automatically loaded into in the editor. The expected result of such a double click on list3_7.lua is shown in Figure B.2. The left side of the figure shows the code from the list3_7.lua file. It is readily seen that the editor has syntax highlighting for Lua reserved words and other

language structures. The right side shows the output from executing the Lua file. Program execution is readily performed from the SciTE editor from the Tools pull down menu. Right clicking on the Tools button along the top of the SciTE editor will give a drop down menu with the options of "compile" and "go". Selecting the "compile" option will execute the Lua compiler which provides a check of the program syntax without executing the code. The "go" option will compile and execute the Lua program and show any generated output from the program in the right half screen

.

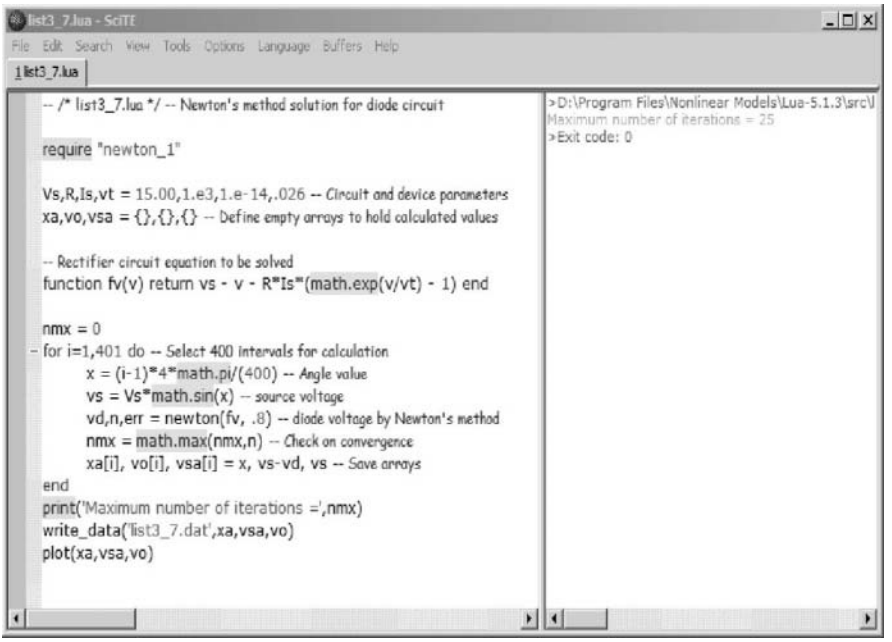

Figure B.2. Lua file list3_7.lua listed in the SciTE editor. Left side shows the Lua code file and the right side shows the result of executing the Lua file.

The Lua code on the left may be readily changed in the SciTE editor and the program re-executed without leaving the SciTE editor. This is the recommended way for execution the example code in this book.

If the software has been properly downloaded into the disk directories as indicated by Figure A.1 but a double click of the mouse on a *.lua file does not bring up the SciTE editor with the loaded code, it may be necessary to manual configure the software to associate the SciTE editor with a *.lua file. This can be performed as follows. Right click on any of the *.lua files to bring up a menu with an entry entitled "open with". Select this item to bring up the "Open With" menu and from this use the "Browse" button to select the SciTE.exe file in the Nonlinear Models\SciTE174\wscite directory. Also be sure to check the "Always use the selected program to open this kind of file" button. A double click on any *.lua file should then result in the *.lua file being loaded into the SciTE editor.

Of course any editor can be used to edit a *.lua file as it is just a pure text file. However, many simple text editors do not have syntax highlighting and automatic program execution. In any case one should use a programming language oriented editor with syntax highlighting and program execution mode such as the SciTE editor.

Assuming that the demonstration suggested above executes properly the download of the software has been successful and all the examples in the book should be easy to execute.

B.3 Compact Installation of Software

The "compact" software installation downloads all the software associated with the Nonlinear Models directory on the disk but does not download any of the Examples code for the book. This decreases the disk space required from about 120MB to about 35MB of disk space. In order to execute the Lua programs discussed in the book it is then necessary to separately download some of the chapter examples from the Examples directory. However, these can be downloaded on a chapter by chapter basis thereby saving disk space. If the user has sufficient disk space it is highly recommended that all the chapter examples simply be downloaded with the "full" installation. If the "compact" installation is used, and the download is successfully completed, the computer should be configured so that a double click on a *.lua file will properly load the file into the SciTE editor for changes or for execution. The reader is referred to the previous section for a discussion of the use of the software.

B.4 Manual Installation of Software

Under some circumstances the user may wish to manually install part or all of the supplied software. Cases where this might occur are when the user's computer already has some of the software already on the computer – for example already has Lua or SciTE on the computer. In this case the following procedure can be used to download selected parts of the supplied software. However, if sufficient space is available on the user's hard drive, it is highly recommended that the user simply download the extra copies of the software since the Lua examples are properly configured for use with software in the indicated directories. To download all of the supplied programs and examples requires about 120MB of disk space (or about 35MB if the examples are not downloaded). Downloading all the programs should not interfere with the user's execution of any previous software already on the user's computer, even if the user already has Lua, gnuplot or SciTE software on his/her computer. The following instructions are for the user who insists on downloading only selected parts of the software.

The user should select a disk drive and directory for storing the downloaded software. The default assumed directory is C:\Program Files\Nonlinear Models.

The user can select any other desired directory and for discussion purposes this directory will be referred to as the TopDir. The user can simply copy any of the programs on the software disk into this TopDir, such as the EasyMesh and gnuplot directories. For reference a brief description of the software in the supplied directories is:

EasyMesh – Programs used in Chapter 13 for generating a triangular mesh.
gnuplot – Programs for generating pop-up graphs of functions – used in all chapters.
Lua-5.1.3 – Lua software for programming language used in book
SciTE174 – Programming text editor recommended for use with Lua

These directories and associated files can simply be copied to the TopDir of the user's computer.

For proper execution of the Lua programs in the Examples directory the Lua programs in the Nonlinear Models\Lua-5.1.3\src\lua directory on the disk must be accessible by the lua.exe program. If the user already has a version of Lua installed on his/her computer these files must be located in the src\lua directory of the user's Lua language directory. An experienced user of Lua should know how to achieve this. In any case even if the user has a copy of Lua already on his/her computer there is no harm in having another version of Lua in another directory if sufficient disk space is available (the Lua files on the disk require only 1.8MB of disk space). It is thus highly recommended that the user simply download the Lua files from the disk even if he/she already has Lua installed on his/her computer.

For a manual download of the files some configuration of the files is required for proper execution of the programs for each chapter. If the user selects TopDir = C:\Program Files\Nonlinear Models as the directory for the software, the downloaded programs already are configured for this directory and no modifications of the programs are required. The user can then skip the remainder of this section. Otherwise several programs must be modified so the software can properly locate needed files. First the software must know where (in what directory) the gnuplot software is located in order to produce the pop-up graphs used in the example programs. The program that must be modified is the lua.lua file in the "Lua-5.1.3\src\lua" directory. On the fourth line of this file should be the statement:

```
local TopDir = "C:\\Program Files\\Nonlinear Models\\"
```

Using a text editor this statement should be modified to describe the directory into which the gnuplot software is located (yes the double \\ are required).

Second a file associated with the SciTE editor must be modified to properly tell the editor where the Lua executable code files are located. The file that must be modified is the lua.properties file in the "SciTE174\wscite" directory. Again using a text editor for this file, near the end of the file should be the statements:

```
command.compile.*.lua=C:\Program Files\Nonlinear
   Models\Lua-5.1.3\src\luac.exe -l lua.lua
   -o "$(FileName).luc" "$(FileNameExt)"
# Lua 5.0
command.go.*.lua=C:\Program Files\Nonlinear
   Models\Lua-5.1.3\src\lua.exe -l lua.lua
   "$(FileNameExt)"
```

The "C:\Program Files\Nonlinear Models" text must be changed to the directory in which the Lua files are located.

If the disk software is manually downloaded to the user's hard drive the above modifications must be manually made only if the user downloads the software to a directory different from the default of "C:\Program Files\Nonlinear Models". With the changes indicated above the software should be properly configured for executing the example programs. The final modification would be configuring the computer so that a double click on a *.lua file automatically loads the file into the SciTE editor. If the user is not proficient in this operation, the following procedure will perform the task. Right click on any of the *.lua files to bring up a menu with an entry entitled "open with". Select this item to bring up the "Open With" menu and from this use the "Browse" button to select the SciTE.exe file in the appropriate SciTE174\wscite directory. Also be sure to check the "Always use the selected program to open this kind of file" button. A double click on any *.lua file should then result in the *.lua file being loaded into the SciTE editor.

Subject Index