

Programming for Evolutionary Biologists

John P. Huelsenbeck

*Department of Integrative Biology
University of California, Berkeley*

May 28, 2022

Contents

1	Introduction to Programming	1
1.1	On becoming a card-carrying computational biologist	1
1.2	What is programming?	2
1.3	Choosing a programming language	2
1.4	Goals	3
2	IDE Misery	5
2.1	What is an IDE?	5
2.2	Getting started with Xcode	6
2.3	Getting started with CLion	8
2.4	Getting started with Eclipse	8
3	Hello World: the World's Most Boring Program	9
3.1	Hello World: Boring, but informative	9
3.2	<code>main</code> is a function	9
3.3	Every C++, and C, program starts with <code>main</code>	10
3.4	<code>#include</code> statements expose the built in functions of the language	10
3.5	Please explain <code>std::cout << "Hello, World!" << std::endl</code>	12
3.6	Functions take arguments and return values	13
4	Variables are Fun!	17
4.1	The basic variable types	17
4.2	Variables take up space	19
4.3	Variables in computers have limits	20
4.4	You can use a variable to hold a memory address	22
4.5	Dereferencing pointers	24
4.6	Arrays	25
5	Conjunction Function	31
5.1	Making your own functions	31
5.2	You can pass variables to a function by value or by reference	32
5.3	Variables have a scope	37

6	A Pseudorandom Number Class	41
6.1	Into the deep end of the pool	41
6.2	Making your first class	42
6.3	Why do we split the class into .hpp and .cpp files?	43
6.4	Instantiating a class	45
6.5	Implementing the uniformRv function	51
6.6	Implementing the exponentialRv function	53
7	Markov chain Monte Carlo	55
7.1	Some theory	55
7.2	MCMC	59
7.3	Coding MCMC for coin tossing	60
7.4	Summarizing MCMC output	68
8	Representing Trees in Computer Memory	73
8.1	The basic idea	73
8.2	The <code>Node</code> class	74
8.3	The <code>Tree</code> class	77
8.4	Traversing trees in pre- or postorder	84
9	Simulating the Birth-Death Process of Cladogenesis	89
9.1	A stochastic model of cladogenesis	89
9.2	Coding the birth-death process	90
9.3	Printing the tree as a Newick string	103
10	Simulating Sequence Evolution on a Tree	109
10.1	Assumptions of phylogenetic methods	109
10.1.1	Transition probabilities	111
10.1.2	Stationary distribution	114
10.2	Four equivalent ways to simulate DNA sequences on a tree	117
10.2.1	Method 1	119
10.2.2	Method 2	120
10.2.3	Method 3	120
10.2.4	Method 4	121
10.3	A C++ simulator	121
10.3.1	Modifying the <code>Tree</code> class	122
10.3.2	Creating the <code>Alignment</code> class	131
10.3.3	Using PAUP* to analyze the data	146
10.3.4	How well do alternative methods work?	146
Appendices		
Appendix A Random Variable Class		151
Appendix B MCMC Coin Tossing Program		155

<i>CONTENTS</i>	1
Appendix C A Simple Tree	159
Appendix D Birth-Death Process of Cladogenesis Code	165
Appendix E Site Probabilities	173
Bibliography	175

Chapter 1

Introduction to Programming

1.1 On becoming a card-carrying computational biologist

As an evolutionary biologist interested in programming, you are joining a group with a proud history that extends back to the beginnings of the computer era. It is largely unacknowledged that the field of computational biology was founded by biologists, and not just by any flavor of biologist, but by *evolutionary* biologists. In fact, Sir Ronald A. Fisher — yes, *that* R.A. Fisher who was a founder of modern population genetics as well as an architect of modern frequentist statistics — is likely the first person to have used a computer to solve a problem in biology. Fisher (1950) used an EDSAC computer in Cambridge to compute the expected allele frequencies in a cline. Just a little more than a decade later, in 1963, his former postdoc and student, Luca Cavalli-Sforza and Anthony Edwards, respectively, wrote some of the earliest programs to estimate phylogeny, executing the programs on an Italian-made Olivetti Elea 6001 computer (Edwards, 2009). The next generation of evolutionary computational biologists came of age, scientifically at least, in the 1970s and 1980s and included such luminaries as Joe Felsenstein, Elizabeth Thompson, Steve Farris, Masatoshi Nei, David Swofford, and the brothers, Wayne and David Maddison. They addressed all sorts of problems in evolutionary biology, but mainly concentrated in population genetics and phylogeny estimation which posed problems that could not be solved analytically, but were amenable to numerical solution using a computer. Although these researchers did not necessarily all get along, it was at this time that a community consisting of evolutionary biologists who took a computational approach in their research developed. As a rule, this group of researchers was generous in sharing their knowledge with others; they passed on their knowledge, either through formal mentorship or through more informal means (which typically involved some consumption of beer), to the next generation which includes too many biologists to list here. This author, in fact, learned many of the tricks of the trade for dealing with trees in computer memory from David Swofford¹. This book is my attempt to pay it forward by passing some of the tricks of the trade on to you, a member of the next generation of evolutionary computational biologists. My hope is that my means of paying it forward, in the form of this book, will be as instructive to you as David Swofford's tutelage was for me.

¹Swofford is a strong proponent of the informal method for passing on programming knowledge.

1.2 What is programming?

Programming is the act of writing instructions for a computer to perform. The instructions are written in a language, called a programming language of which there are many to choose from. Typically, the instructions are simply written on a plain text file which is then read by a computer program called a ‘compiler’ which compiles the code, turning your instructions into a form that can be read directly by the computer. The coding language is a necessity as it is readable by ordinary humans whereas the compiled code, which is gobbled up by the computer and executed instruction-by-instruction, is much more difficult to comprehend. The coding language also helps ensure portability of the ideas expressed in your code to various computer platforms. Machine-readable code — the output of a compiler — is machine specific, so the executable code for one computer and operating system will not necessarily work on another. The code, however, can be ported from computer to computer and compiled for each.

1.3 Choosing a programming language

Programming is difficult. Turning a concept into an algorithm that is implemented in code is not an easy task, for one. But, for the beginner, there are several hurdles that must be cleared before the interesting problem of algorithm development can even be tackled. You need to master, or begin the process of mastering, the programs that allow you to program. I will discuss these programs, called Integrated Developer Environments (IDEs), in the next chapter. The first decision to make, however, is which programming language to use. All programming languages have similarities, and as you become more experienced, you will notice these similarities and realize that learning two different computer languages, such as C++ and Fortran, is much easier than learning two spoken languages such as Spanish and German. But, these similarities won’t become apparent until you have one language, at least, under your belt. Which language should you invest your precious time in learning?

Popular computer languages include Java, Python, JavaScript, C, C++, C#, PHP, Perl, Swift, R, Rust, and Ruby, to name a few. More experienced programmers can be helpful in guiding you in choosing a first language, but you should beware. Programmers often have very strong opinions about the merits (and demerits) of various languages. In this book I chose to use C++ to illustrate ideas. C++ is not a bad choice for a first language to learn. You could do worse. It is widely used and powerful.

Importantly, C++ is a compiled language. “What is a compiled language?,” you ask. A compiled language is one that must be run through a compiler, which produces instructions specific to the target machine. Some languages, such as the popular Python, are interpreted languages. The instructions for an interpreted language are not directly executed by the target machine, but rather are read and executed by another program, cleverly called the ‘Interpreter.’ Interpreted languages are often easier to run and use, but typically do not run as fast as compiled languages.

A C++ programmer can also take advantage of the code written by others to solve various problems, an advantage C++ shares with other languages. This code is referred to as a library. Sometimes, they are precompiled. The important point is this: you can integrate the library into your program and use the functionality of the library to solve your own problems. This saves you a lot of time, allowing you to concentrate on the aspects of your problem that are truly unique. Moreover, libraries can often be of very high quality. The routines you take advantage of in a library

are likely better-written than anything you (or I) could write. An example of a C++ library often used by scientists is the Boost Library.

1.4 Goals

This book is not meant to provide you with an in depth review of C++ and programming. Rather, my goal is to get you started with programming in evolutionary biology. The exercises presented in each chapter build on each other. The idea is that by working through the exercises, you will not only learn a lot about programming, but better understand many of the algorithms that are used by evolutionary biologists.

As I mentioned, above, programming is hard. You should not expect to understand every concept on the first go. Be persistent! Don't give up! If you approach programming with a positive attitude, you'll find yourself enjoying the process.

Good luck and happy programming!

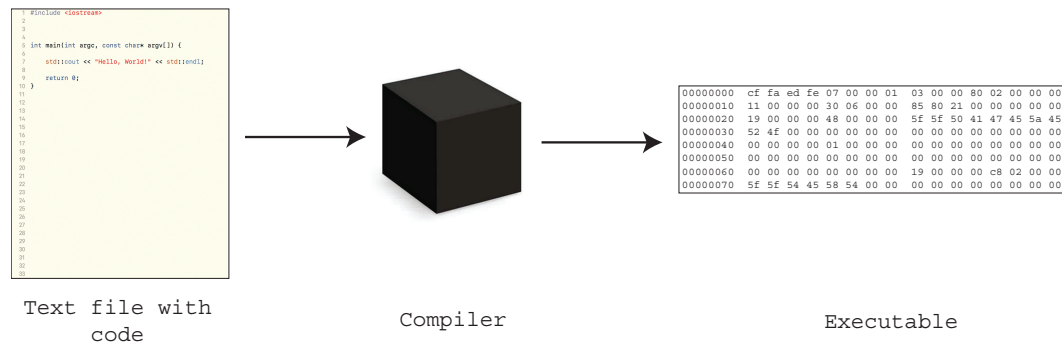
Chapter 2

IDE Misery

2.1 What is an IDE?

When I was a graduate student, my advisor bought me a powerful (for the time) and exceedingly expensive Sun SPARC workstation. It was the first computer I had used that ran a Unix operating system. For the first two weeks, until I figured out how to install and operate the compiler, this workstation was nothing more than an expensive time piece. The process of working out how to install and use the compiler was, for me, a frustrating experience. However, my experience was not unique; the first, and worst, experience for a beginning programmer is learning how to actually compile and run a program for the first time. Fortunately, it is much easier to get started programming today. Why? Integrated Developer Environments (IDEs).

Remember that writing, compiling, and running a program involve the following steps:



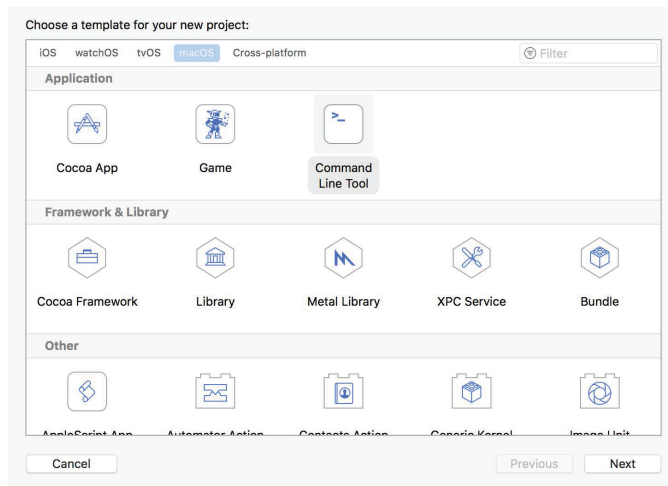
Entering the code into a text file can be accomplished using any text editor, even Microsoft Word, as long as the file is saved as a plain text file. And, compiling the code can be performed directly, by running the compiler from the command line. IDEs, however, simplify both tasks by wrapping the code generation and compilation into one program. The text editor for an IDE can be customized to suit your style, or the style of your programming team. Colors can be used to indicate different aspects of the code, such as control statements, variables, and comments. The IDE also manages the source code files, of which there may be many, that constitute the program. The compiler is run from the IDE. Typically, the settings for the compiler can be set from the IDE. Finally, IDEs have built in debuggers, which can help you track down errors in your code.

There are many IDEs that you can use. In this book, I will assume that you are using Xcode, if you are using a computer running the MacOS. If you are using a computer running the Windows operating system, I would recommend installing CLion, Eclipse, or Visual Studio.

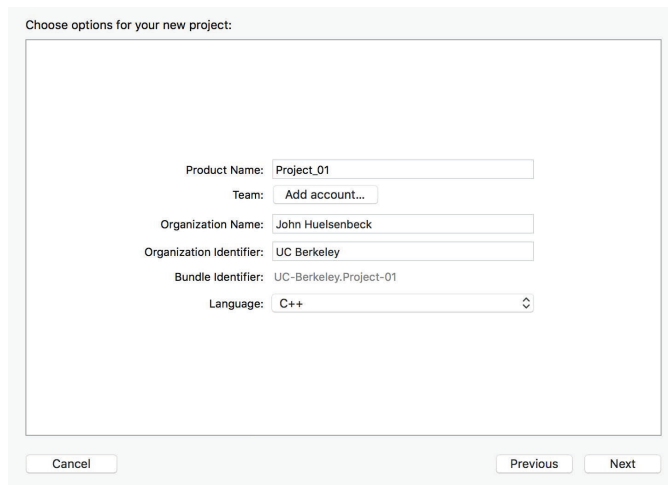
2.2 Getting started with Xcode

XCode is an IDE developed by Apple and given away for free, which is a pretty good deal in my opinion. You can download XCode through the App Store application on your Mac. Search for XCode in the App Store search field, then click on the ‘Download’ button. XCode is a large program, so be patient while the program downloads to your computer.

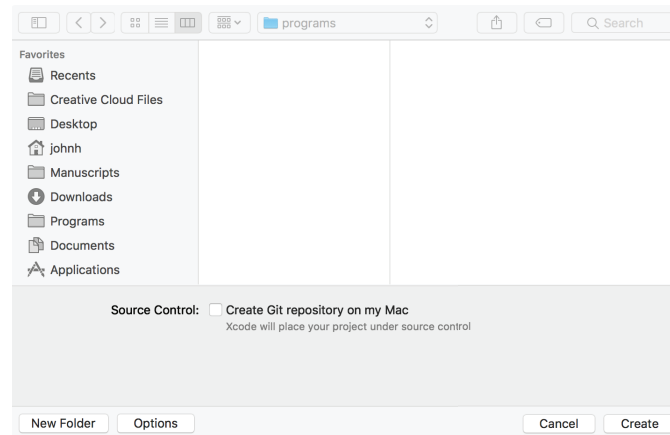
Launch XCode after you download it. Select **New > Project** from the **File** Menu. You will see the following window:



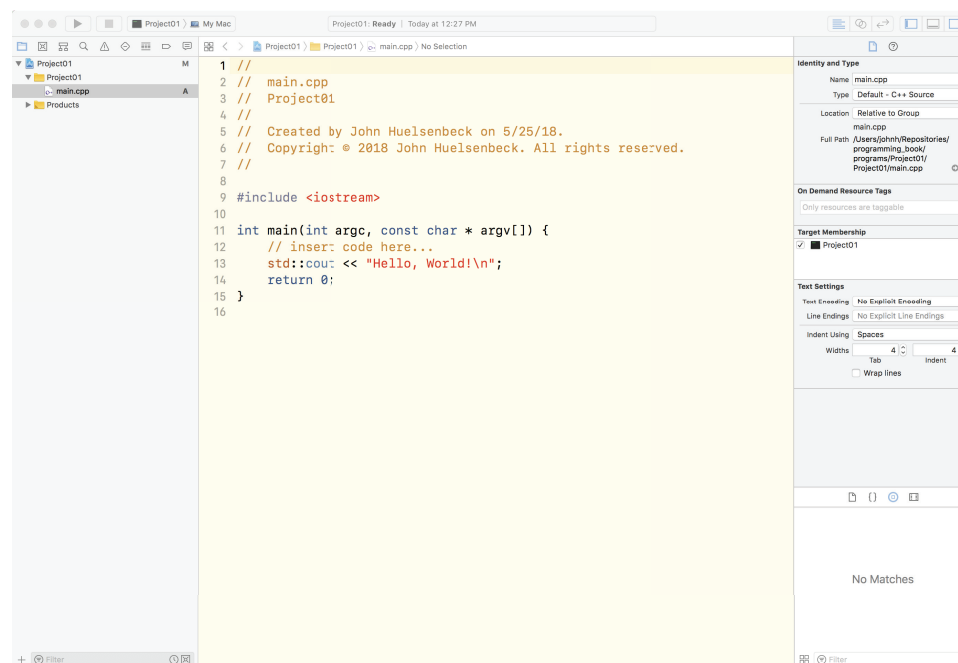
Select **Command Line Tool** and click on **Next**. This leads to the following window:




Name the project. Here, I cleverly named the project **Project01**. You can name your project whatever you like. However, I will be referring to this program as ‘Project01’ throughout this book, so you can simplify your life by following my lead here. You might consider entering your name instead of my name in the **Organization Name** field. Similarly, enter something sensible for **Organization Identifier**. Most importantly, make certain that **C++** is selected as the **Language**. Then select **Next** to continue to the next (and, thankfully, last) window that allows you to save your project to a specific location on your computer:

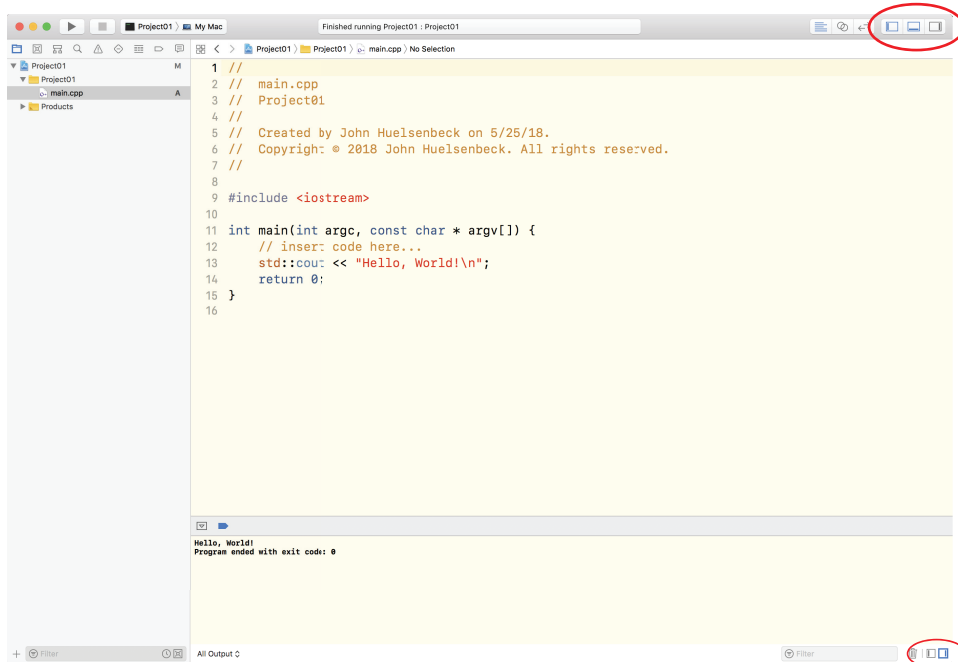


Do not choose to **Create Git repository on my Mac**. I'll explain source code repositories, such as Git, in a later chapter. You should see the following after you save your project:



Congratulations! You have just ‘written’ your first computer program. How? XCode fills in a

bit of starter code for you. If you want to, you can compile and run this code by clicking on the ‘play’ button () in the top-left corner of the window.



Play with the toggles, indicated by the red ellipses until you get the window to look like the one, above. The window, above, contains just the information you need to write the code (the portion in the top window) and see the results of the program running (the portion in the lower window).

2.3 Getting started with CLion

2.4 Getting started with Eclipse

Chapter 3

Hello World: the World's Most Boring Program

3.1 Hello World: Boring, but informative

If your IDE has not already done so, enter the following text in the `main.cpp` file for the first program you set up in your IDE, called ‘Program01’:

Example

```
#include <iostream>

int main(int argc, char* argv[]) {

    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

The code you have entered implements the well-known ‘Hello, World’ program. Almost every introductory programming book starts with some variant of this program. What does the program do? It simply prints the phrase, “Hello, World!” to the standard output on your computer. (In Xcode, the output is directed to a window in the lower portion of the viewer. In Eclipse, the standard output is directed to ...) Clearly, this is a very boring program. You would think we could do better. (We can.) That said, the Hello World program illustrates a few points.

3.2 `main` is a function

Mathematicians use functions all of the time. For example, consider the function called $f(x)$:

$$f(x) = x^2$$

This function takes as input the variable x and outputs the square of x , or x^2 . Programmers also make extensive use of functions. In fact, the Hello World program has one function, called `main`. The outline of the `main` function is:

```
int main(int argc, char* argv[]) {

}
```

The `main` function takes as input arguments two variables which are listed in the parentheses (`int argc`, `char* argv[]`, which will be discussed in more detail later). The `main` function also returns a value, the type of which is an integer (called `int` here). (Note, we will discuss variables extensively in the next chapter.) The body of the function is the space between the two curly brackets.

How would we define a function that squares a value, like the simple example function we gave above in which $f(x) = x^2$? The following would accomplish this:

```
double squareFunction(double x) {

    return x * x;
}
```

The function is named `squareFunction`. The function takes as an argument the value `double x`. What does this mean? The word `double` refers to the variable type. We won't go into great detail about `double` variables here, but for now it suffices that `double` variables can hold real numbers, such as 3.14. The function also returns a value, which is the square of the value that is passed into the function as an argument. (Here, `x * x` is equivalent to $x \times x$.)

3.3 Every C++, and C, program starts with `main`

You now have a basic idea of how functions are defined in C++. The pattern is

```
<return type> <function name>(<input variable(s)>) {

}
```

with the body of the function being the commands that are typed between the curly brackets.

Every C++ program begins execution at a function called `main`. Every C++ program must contain one, and only one, `main` function. If you ever happen to be examining someone else's code, look for the `main` function. Everything starts there.

3.4 `#include` statements expose the built in functions of the language

Change line 1 of the program by typing two backslashes in front of the statement, `#include <iostream>`. The line should look like

3.4. #INCLUDE STATEMENTS EXPOSE THE BUILT IN FUNCTIONS OF THE LANGUAGE11

Comment out the header:

```
//#include <iostream>
```

The double slash denotes comments in your code. Everything after the `//` on that line is not code that is read by the compiler, but only there to help you and others understand the code. So, the `//` in front of the `#include` ‘comments out’ that line of code. What happened when you did this? In Xcode, I get the following error pop up in the IDE, next to that line: ‘Use of undeclared identifier ‘std’.’ What is the include statement doing that is so important that leaving it out breaks the program?

Any command that starts with the pound sign, `#`, is called a ‘compiler directive.’ The compiler directive is read by the preprocessor of the compiler, which reads the code before compiling, resolving any of the directives. The `#include <iostream>` directive tells the compiler to look for a file called `iostream` and include that file in the project.

On my computer, `iostream` is a physical file located in the Xcode program bundle. To understand what is going on here, I’m going to back up a few steps and give you a more general view of a programming language. Imagine the following scenario: two different companies are writing C++ compilers. Company 1 decides that they really don’t like how C++ uses the word `double` when referring to a variable that will hold a real number. Sensibly enough, they decide that in their compiler, whenever you want to declare a variable that will hold a real number, you will type

```
real x;
```

which would declare a variable called `x` to be of type `real`. Meanwhile, the programming team from Company 2, which happens to be German, decides like Company 1 that `double` is not a sensible name for a real number. But, being German, they decide that in their compiler program variables that hold real numbers will be declared

```
zahl x;
```

Both companies have made seemingly sensible decisions. What could go wrong?

It turns out that a lot could go wrong if compiler coders made arbitrary decisions like the ones I described. Imagine that you wrote a computer program in C++ that is read by, and compiled on, Company 1’s compiler. You give your code to your friend, who happens to have bought Company 2’s compiler. Your friend would not be able to compile your code without the hassle of changing all of the ‘reals’ to ‘zahls.’ Yikes! Or, I should say, “Heiliger Strohsack!”

To prevent such chaos, writers of compilers adhere to a standardized version of the C++ language. The language standards, of course, are maintained by a committee, in this case the International Organization for Standardization (ISO). A C++ compiler must closely adhere to the ISO guidelines in order to be ISO compliant. Variables that store real numbers must be called `double`, not `real` or `zahl`. And, certain built-in functions must not only be implemented, but must be implemented in certain files that can be included to expose that functionality. The line that begins `std::cout` outputs text to the console. `std::cout` is implemented in the `iostream` file, which must be included if you use `std::cout` in your code.

Several standards for the C++ language have been released by ISO. The most recent is C++17, which was released in December 2017.

Throughout this book, you will see examples of new functions that require different files to be included (*e.g.*, if you want to take the natural log of a number, using the `log` function, you need to include the `cmath` file). Why didn't the C++ standard just include *all* of the functions automatically? Why include functionality piecemeal? There are several reasons. For one, by including only those files necessary for the bit of code you are writing, compiling that code is faster. More generally, good programmers follow the principle of including only the minimum necessary to compile code. When you garden, you only bring the tool(s) necessary to prune your shrubs, or weed. You don't bring the entire contents of your gardening shed. Similarly, in programming, we only include the tools necessary for the portion of code expressed in the file.

You will see two examples of `#include` directives:

```
#include <file>
#include "file"
```

The first is used to expose the built-in functionality of the C++ language. Files that you will often include are `iostream`, `iomanip`, `cmath`, `time`, and `fstream`, among others. The second include, using the quotation marks, is used to include files from your program project. You will see that you do not write all of the code in a single text file. Rather, the code that constitutes your program is scattered among files. A large program, such as RevBayes (Höhna et al., 2016), can include thousands of files!

3.5 Please explain `std::cout << "Hello, World!" << std::endl`

The main body of the 'Hello, World' program does two things: print the text "Hello, World!" and then return the number 0 to the operating system. Output is handled by `cout`, which stands for 'console out.' The command, `std::cout` opens an output stream, directed to the console. A stream is an abstraction used for input and output in C++.

C++ uses the abstraction of a 'stream' to control input and output to a device. The input and output can be from the console or to a file or to some object. A stream is nothing but a sequence of characters. One can insert characters into the stream with the `<<` operator. One can also extract characters from the stream with the `>>` operator.

The command, `std::cout` opens a stream to the console. The next command, `<< "Hello, World!"` inserts into the stream the string, 'Hello, World!' A string is a series of characters, that can include spaces. A string is defined by the characters between the quotation marks. The last command, `<< std::endl` inserts into the stream an end-of-line character. (If you are familiar with those ancient devices called typewriters, you can think of the end-of-line character as being a carriage return.)

A lot more could be said about streams. They provide the C++ language with a flexible way to output data and to take in data that is device independent (*i.e.*, the C++ interface remains the same regardless of where the stream comes from or goes to). We'll see more of streams in later chapters.

3.6 Functions take arguments and return values

Earlier, I mentioned that `main` is a function and that the general pattern of functions is:

```
<return type> <function name>(<input variable(s)>) {  
  
}
```

Let's delve more deeply into what the `main` function is taking in as arguments and what it is returning.

The `main` function takes as arguments two variables, `int argc` and `char* argv[]`. Where do these arguments come from? In this case, `argc` and `argv` are supplied by the operating system when the program is executed. `int argc` is a variable called '`argc`' which can hold an integer value. The other variable that is provided by the operating system, '`char* argv[]`,' is a bit more mysterious. This is an array of pointers to strings. I realize that probably made no sense, but here is another try: `char* argv[]` is a vector of memory addresses, each of which indicates the starting address for a string.

What is contained in `int argc` and `char* argv[]`? `int argc` holds the number of strings contained in `argv`. Rewrite your `main` function to look like this:

Example

```
int main(int argc, char* argv[]) {  
  
    std::cout << "argc = " << argc << std::endl;  
    for (int i=0; i<argc; i++)  
    {  
        std::cout << "argv[" << i << "] = " << argv[i] << std::endl;  
    }  
  
    return 0;  
}
```

When I run the above program, I get the following output:

```
argc = 1  
argv[0] = /Users/johnh/Project01/DerivedData/Project01/Build/Products/Debug/Project01  
Program ended with exit code: 0
```

The operating system is passing to the program, through the `main` function, information on how the program was called. In this case, there is one argument and that argument is the path to the executable.

On my computer, a Macintosh, I opened the `terminal` app and typed the path to the executable, which is everything in the line, above, except the executable name:

```
cd /Users/johnh/Project01/DerivedData/Project01/Build/Products/Debug/
```

If I type the unix command, `ls`, I see the list of files in the `Debug` directory: ‘Project01.’ The executable can be run using the following command,

Type the command:

```
./Project01
```

which gives the following output:

```
argc = 1
argv[0] = ./Project01
```

Now, let’s have some fun. I am going to type the following the next time I execute the program,

Type the command:

```
./Project01 David Swofford had a little lamb which was named, sensibly enough, PAUP
```

This line produces the following output on my computer:

```
argc = 13
argv[0] = ./Project01
argv[1] = David
argv[2] = Swofford
argv[3] = had
argv[4] = a
argv[5] = little
argv[6] = lamb
argv[7] = which
argv[8] = was
argv[9] = named,
argv[10] = sensibly
argv[11] = enough,
argv[12] = PAUP
```

Note that the operating system broke up the sentence by words, separated by blank space(s). Again, the first string that is passed to the `main` function is the path to the executable name. The other strings, however, are the individual words that followed the executable name.

You may have had experience with programs that run from the command line. Genomics analysis often involves the use of many command-line programs, perhaps stitched together using a language such as python. For example, ClustalW is a widely-used program that aligns nucleotide or amino acid sequences. The user manual gives the following example of how to call ClustalW from the command line:

```
clustalw2 -infile=my_data -type=protein -matrix=pam -outfile=my_aln -outorder=input
```

Note that the word `clustalw2` is the executable name. Simply typing this word executes the program. All of the words after `clustalw2`, however, are arguments that are passed into Clustal's main function. The programmers for Clustal read the number of arguments (`argc`) and the strings `argv` to set the program's state. Command line arguments passed into `main` using `int argc` and `char* argv[]` are a convenient, if not particularly user-friendly, way to specify things such as input and output files, *etc.*

Finally, the program returns to the operating system the number 0. It does this using the return statement, `return 0`.

This concludes our discussion of the 'Hello World' program. Who would have thought that such a simple program would provide so much fodder for discussion?

Chapter 4

Variables are Fun!

4.1 The basic variable types

Modify the main function for your first project, `Project01` so that it reads:

Example

```
int main(int argc, char* argv[]) {  
  
    int x = 3;  
    std::cout << "x = " << x << std::endl;  
  
    return 0;  
}
```

This simple program declares a variable named ‘x’ and initializes its value to `x = 3`. This all occurs on one line, `int x = 3`. The next line prints the value of x. The output of the program should look like:

```
x = 3  
Program ended with exit code: 0
```

The single line in which we declared and initialized the variable x could have been done in two lines, instead:

```
int x;    // declare variable called x  
x = 3;    // set the value of the variable x to 3
```

Most programmers, when declaring a variable, will also initialize it to some value. Normally, if I were declaring a variable like x, I would initialize its value to zero, `int x = 0`, so that I could rely on it being zero. Some compilers, when you declare a variable without initializing it (*e.g.*, `int x`),

initialize the value to zero. Other compilers, however, do not do this and the value of the variable will reflect the pattern of bits that happen to be at that memory address. Declaring and initializing the variable ensures that it has a value that you can rely on. Note that I added comments to the two lines, above; the words after the `//` are the comment and are not read by the compiler.

In C++, when you declare a variable, you also indicate the variable type. Here, we are declaring the variable `x` to be of type `int`. In C++, and other languages too, `int` declares a variable that can hold integers. (Remember, integers are the numbers $\dots, -3, -2, -1, 0, 1, 2, 3, \dots$)

What if you wanted to store and manipulate a real-valued number in computer memory? You can do this with the `float` or `double` variable types. Modify the `main` program, again, to read:

Example

```
int main(int argc, char* argv[]) {

    int x = 3;
    std::cout << "x = " << x << std::endl;
    double y = 3.14;
    std::cout << "y = " << y << std::endl;

    return 0;
}
```

When I run this program, I get the following output:

```
x = 3
y = 3.14
Program ended with exit code: 0
```

The other standard variable types are summarized in the following table:

Variable	Type	Example
<code>int</code>	Integers	-1, -100, 0, 314, 10001
<code>unsigned int</code>	Natural Numbers	0, 1, 2, ...
<code>float</code>	Real Numbers	-4.23, 10.01e+4, 10203.0001
<code>double</code>	Real Numbers	Same as with floats
<code>char</code>	Characters	c, a, B, Z
<code>bool</code>	Boolean	true, false

There are a few other variable types that you might come across, but the table summarizes the main ones you will likely ever use.

4.2 Variables take up space

When you declare a variable, such as `int x = 0`, the operating system sets aside enough space on your computer's memory to represent the variable. Interestingly, C++ allows you to see how much space is set aside and even where the variable resides in memory. Rewrite `main` in `Project01` to read:

Example

```
int main(int argc, char* argv[]) {

    int x = 0;
    std::cout << "x's value    = " << x << std::endl;
    std::cout << "x's address = " << &x << std::endl;
    std::cout << "x's size   = " << sizeof(int) << std::endl;

    return 0;
}
```

When I run this program, I get the following output:

```
x's value    = 0
x's address = 0x7ffeefbfff57c
x's size     = 4
Program ended with exit code: 0
```

The first line of code, `int x = 0`, simply declares and initializes an integer variable called `x`. The next line of code should also make sense to you; it simply prints out the value of `x`, which because you initialized the variable at the same time that you declared it, is predictably zero. It's on the next line of code which prints out the memory address of `x`, `std::cout << "x's address = " << &x << std::endl`, where things get interesting. You can get the memory address of a variable by putting the ampersand symbol, '&,' in front of the variable's name. So, `&x` represents the memory address of the variable `x`. You can see that when I ran the program on my computer, the memory address is reported to be `0x7ffeefbfff57c`.

The memory address deserves explanation. First of all, the memory on your computer is arrayed in bytes, each of which has an address. The address is either a 32 bit or 64 bit number. On my computer, the addresses are 64 bits and output as hexadecimal numbers. The number representing the memory address for `x`, `0x7ffeefbfff57c`, is in hexadecimal format¹ It's unlikely that when you

¹Converting between number bases is not too difficult. Remember that a base-10 number is in the following format: $\dots \times 10^4 + \dots \times 10^3 + \dots \times 10^2 + \dots \times 10^1 + \dots \times 10^0 + \dots \times 10^{-1} \dots$ where the blanks are filled with the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, or, 9 depending on the number that is represented. For example, the number 1066 would be $1 \times 10^3 + 0 \times 10^2 + 6 \times 10^1 + 6 \times 10^0$. A binary (base two) number works in a similar fashion, but the format is $\dots \times 2^4 + \dots \times 2^3 + \dots \times 2^2 + \dots \times 2^1 + \dots \times 2^0 + \dots \times 2^{-1} \dots$, with the blanks filled in with the digits 0 or 1. A hexadecimal

ran the program that your computer put the value `x` in the same place in memory as when I ran the program on my computer.

The last line of the modified program prints the size of the variable, `x`. You can see that an integer takes up four bytes on my computer. In fact, the first byte will reside at the address that was printed out (`0x7ffeefbff57c`). The next three bytes will be adjacent to the first, at the locations `0x7ffeefbff57d`, `0x7ffeefbff57e`, and `0x7ffeefbff57f`.

Of course, the memory address of a variable can change each time the program is run. The amount of space that is set aside for each variable type, however, is constant. On my computer, the standard variable types take up the following amount of space:

Variable	Number Bytes
<code>int</code>	4
<code>unsigned int</code>	4
<code>float</code>	4
<code>double</code>	8
<code>char</code>	1
<code>bool</code>	1

4.3 Variables in computers have limits

We now know that when we declare a variable to be of type `int`, that the compiler sets aside four bytes of space somewhere on your memory card. What is the largest and smallest value that can be stored in computer memory as an `int`?

Each byte of memory consists of eight ‘bits,’ each of which has two states, on (1) or off (0). The binary representation of the first ten positive integers is

Number	Binary Representation
0	00000000000000000000000000000000
1	00000000000000000000000000000001
2	00000000000000000000000000000010
3	00000000000000000000000000000011
4	00000000000000000000000000000100
5	00000000000000000000000000000101
6	00000000000000000000000000000110
7	00000000000000000000000000000111
8	00000000000000000000000000001000
9	00000000000000000000000000001001
10	00000000000000000000000000001010

Note that most of the bits, the leading ones, are not being used for our `int` variable. In fact, if we knew that the largest number we wanted to hold was 10, we could get away with one byte (8 bits), and still have four bits to spare. There is a variable type called ‘`short int`’ which only takes up two bytes of memory. In this case, in which we know that the maximum size of the integer we want

number is in base 16. The format for the numbers is $\dots \times 16^4 + \dots \times 16^3 + \dots \times 16^2 + \dots \times 16^1 + \dots \times 16^0 + \dots \times 16^{-1} \dots$, with the sixteen digits being 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, *A*, *B*, *C*, *D*, *E*, and *F*, the letters standing for 10, 11, 12, 13, 14, and 15 (in base-10).

to hold is 10 (in base-10), we could use a `short int` instead, thereby saving two bytes of memory. If we were programming in the 1970s, we might go ahead and do just this. In the 1970s, a good computer had only thousands of bytes of memory. Today, we have billions of bytes of memory to play with. Most programmers do not worry about saving a few bytes. Rather, they worry more about places in the code that a lot of memory is allocated (where a lot of memory might be millions of bytes, or megabytes, are allocated).

Imagine we had only two digits to represent a base-10 number. What is the largest value that can be represented with two digits? The answer: 99 (or $10^2 - 1$). Similarly, the largest binary number that can be represented with 32 bits is 11111111111111111111111111111111, or $2^{32} - 1$. For the `int` variable type, however, one of the 32 bits is used to indicate whether the number is positive or negative. Therefore, the largest value that can be stored using an `int` variable is $10^{31} - 1 = 2147483647$ whereas the smallest value is -2147483648 . The actual story is a bit more complicated than I just made out, but you see the point: if we wanted to store a number larger than about 2.1 billion, we are going to run into problems. In fact, if we attempt to do so, we will get an overflow error from the computer. (The opposite problem — attempting to hold the value of a number that is too small — is called an ‘underflow error.’)

Note that you can get information on the limits of numerical representation from functions defined in the `limits` include file.

Similarly, we cannot represent the real numbers with complete accuracy. Try the following experiment; rewrite your project code so that it reads:

Example

```
#include <iomanip>
#include <iostream>

int main(int argc, char* argv[]) {

    float x = 0.1;
    double y = 0.1;

    std::cout << std::fixed << std::setprecision(50) << "x = " << x << std::endl;
    std::cout << std::fixed << std::setprecision(50) << "y = " << y << std::endl;

    return 0;
}
```

The output should look like this:

```
x = 0.100000000149011611938476562500000000000000000000000
y = 0.100000000000000000000000000000000000000000000000000
Program ended with exit code: 0
```

Neither number is exactly equal to 0.1, though both are quite close to that value. The reason is straight-forward: computers cannot represent every real number with complete precision, but rather approximates any particular number as well as it can. The representation of 0.1 is better when we use a `double` variable than when we use the `float` type. This makes sense because the internal representation of the real number for the `double` type uses twice as many bytes as the `float` type.

You should always be concerned about numerical accuracy when doing computations in evolutionary biology. Some of the more obscure output from many programs, such as reporting the log of a probability, are done to avoid underflow.

4.4 You can use a variable to hold a memory address

As I pointed out earlier, we can get the memory address of a variable by putting an ampersand in front of the variable in computer code. This seems to be a neat trick, but otherwise useless. After all, why should we care about the location of the variable in memory? This is especially true because we, as the programmer, do not even control where the variable is to reside.

It turns out that knowing the memory address is quite important. If we know where a variable resides in memory, we can manipulate that variable. Moreover, other parts of your code, such as the functions you code, also have a memory address when the program is executed. If we know where a function resides in computer memory, we can also manipulate it.

What if we want to remember the memory address of a variable? We can make another variable that will hold the memory address. Such a variable is called a ‘pointer’ in the computer science lingo. Rewrite your little program so that the main function now reads:

Example

```
#include <iostream>

int main(int argc, char* argv[]) {

    int x = 0;
    int* xPtr = &x;

    std::cout << "x = " << x << std::endl;
    std::cout << "&x = " << &x << std::endl;
    std::cout << "xPtr = " << xPtr << std::endl;
    std::cout << "&xPtr = " << &xPtr << std::endl;
```

```
std::cout << "int size = " << sizeof(int) << std::endl;
std::cout << "int* size = " << sizeof(int*) << std::endl;

return 0;
}
```

When I run this program on my computer, I get the following output:

```
x = 0
&x = 0x7ffeefbfff56c
xPtr = 0x7ffeefbfff56c
&xPtr = 0x7ffeefbfff560
int size = 4
int* size = 8
Program ended with exit code: 0
```

We declare and initialize two variables in the code. The first should look familiar to you by now. We simply declare a variable called `x` to be of type `int` and set its value to zero (`int x = 0`). The second line is new. Here, we declare a pointer variable of type `int*`. The asterisk indicates that the variable is a pointer. In fact, it is a variable that can hold the memory address of an `int`. We also initialize the pointer variable to be equal to the memory address of `x`.

Confusingly, different programmers will put the asterisk, which indicates that the variable will hold a memory address, in different places. Compilers will accept the following as equivalent:

```
int* xPtr = &x;
int * xPtr = &x;
int *xPtr = &x;
```

It seems the asterisk can attach itself to the variable type (here `int`), to the variable name, or even stay in between the two like a baseball player caught in a pickle. I follow the convention of having the asterisk cling to the variable type.

Two of the lines display the same memory address:

```
&x = 0x7ffeefbfff56c
xPtr = 0x7ffeefbfff56c
```

This makes perfect sense because we set the value of `int*` to be the memory address of `x`. The next line simply shows that the variable named `xPtr` also has a memory address. After all, it is a variable! You will note that the pointer variable takes up eight bytes of memory.

What if, for some reason, we wanted to remember the memory address of the variable `xPtr`? We could do this by declaring another variable to hold the memory address. The big question here is what would the variable type be? Clearly it is a pointer, but it is a pointer to a variable that is itself a pointer. The answer is to append another asterisk to the variable type, resulting in:

```
int** anotherDamnPointer = &xPtr;
```

The variable, `anotherDamnPointer`, can also hold a memory address, but only for variables of type `int*`. You should feel confident enough to modify the program to see that a variable of type `int**` also takes up eight bytes of memory. All pointer variables take up the same amount of memory (four or eight bytes, depending on the computer) regardless of the type of variable it holds the memory address of.

4.5 Dereferencing pointers

I mentioned that if you know the address of a variable, that you can manipulate it. You can do this by ‘dereferencing’ the pointer. Here’s an example using a re-written `main` function:

Example

```
#include <iostream>

int main(int argc, char* argv[]) {

    int x = 0;
    int* xPtr = &x;
    std::cout << "x = " << x << std::endl;

    *xPtr = 3;
    std::cout << "x = " << x << std::endl;

    return 0;
}
```

When I run this program, I get the following output:

```
x = 0
x = 3
Program ended with exit code: 0
```

Note that I didn’t change the value of `x` directly by simply typing `x = 3`. Rather, I changed its value indirectly using `*xPtr = 3`. Essentially, the program changes the value at the memory address stored in the pointer variable, `xPtr`.

4.6 Arrays

Often, a problem can best be solved by using a vector of variables. As an example, in phylogenetics we use vectors of `double` variables to hold conditional probabilities at different points on a tree.

Vectors, called ‘arrays’ in C++ and other languages, can be declared by appending square brackets with the number of variables in the array to the end of the variable name:

```
int x[10];          // declares an array of 10 ints
int* xPtr[10];     // declares an array of 10 int pointers
double y[1000];    // declares an array of 1000 double variables
```

The value for each element can be accessed, again, using the square brackets. Implement the following code in your program:

Example

```
#include <iostream>

int main(int argc, char* argv[]) {

    int x[100];

    for (int i=0; i<100; i++)
        x[i] = i;

    for (int i=0; i<100; i++)
        std::cout << "x[" << i <<"] = " << x[i] << std::endl;

    return 0;
}
```

This little program does three things. First, we declare a vector of 100 `int` variables, called `x`. Second, we initialize each of the 100 variables to be its position in the vector. We do this using a loop. This is the second time you have seen a loop in this book. Here, we use the `for` loop. (There are also `do` and `while` loops, but `for` loops are probably the most frequently used in C++.) The `for` loop has three conditions, separated by semicolons. The first, `int i=0`, declares a counter variable named `i` and initializes its value to zero. The second, `i<100`, indicates the condition for the variable `i`. As long as the condition is met, we will continue through the loop. Here, we will continue through the loop while the variable `i` is less than 100. The third part of the `for` loop increments the value of `i` each time we pass through the loop. The value is incremented at the end of the loop. The statement `i++` is short-hand for `i = i + 1`. Programmers are inherently lazy people. The `++` operator was added to the language to save programmers a few key strokes. The

values of `i` that will be visited in the `for` loop are `0, 1, 2, ..., 99`. The `for` loop could have been replaced by the following 100 lines of code, which you should *not* type:

```
#include <iostream>

int main(int argc, char* argv[]) {

    int x[100];

    x[0] = 0;
    x[1] = 1;
    x[2] = 2;
    x[3] = 3;
    x[4] = 4;
    // 90 lines of similar code!
    x[95] = 95;
    x[96] = 96;
    x[97] = 97;
    x[98] = 98;
    x[99] = 99;

    for (int i=0; i<100; i++)
        std::cout << "x[" << i <<"] = " << x[i] << std::endl;

    return 0;
}
```

Mercifully, I did not write out the full 100 lines of code that would be necessary to accomplish what the two lines of code in the `for` loop accomplished. You can see that loops are really useful!

After initialization, the third and final part of the little C++ program prints out each value of the vector, `x`. It does this using another `for` loop.

The values of the array are adjacent to one another in computer memory. You can see this if you modify the line in the code in which the values are printed to read:

Example

```
#include <iostream>

int main(int argc, char* argv[]) {
```

```
int x[100];

for (int i=0; i<100; i++)
    x[i] = i;

for (int i=0; i<100; i++)
    std::cout << "x[" << i <<"] = " << x[i] << " (" << &x[i] << ")" << std::endl;

return 0;
}
```

When I run the program, the first ten lines of output read:

```
x[0] = 0 (0x7ffeefbff420)
x[1] = 1 (0x7ffeefbff424)
x[2] = 2 (0x7ffeefbff428)
x[3] = 3 (0x7ffeefbff42c)
x[4] = 4 (0x7ffeefbff430)
x[5] = 5 (0x7ffeefbff434)
x[6] = 6 (0x7ffeefbff438)
x[7] = 7 (0x7ffeefbff43c)
x[8] = 8 (0x7ffeefbff440)
x[9] = 9 (0x7ffeefbff444)
x[10] = 10 (0x7ffeefbff448)
```

Note that each `int` variable in the array is four bytes away from the previous variable.

The above example assumes that you know the size of the vector before compiling the program. Here, for whatever reason, we knew we would need 100 integer values in the array. What would you do if you didn't know the size of the array before compiling the program? Perhaps for some input to the program, you will need 50 integer values in the vector but for other input you will need 1000. One possible solution is to simply declare the variable to be of a sufficient size for any possible input. For example, you could write,

```
int x[1000];
```

if you knew that the maximum size of the vector would be 1000 `int` variables. This is an inelegant and inefficient solution to the problem. What if we only need 50 variables? In this case, we would never use 950 of the variables that were set aside when the program is run. The problem is even messier if we cannot predict how many variables we will need when the program is executed, which is the usual case in evolutionary biology in which the size of the problems to be analyzed can vary dramatically.

Fortunately, there is a simple solution. We dynamically allocate the memory that we will need. Rewrite the `main` function to read:

Example

```
#include <iostream>

int main(int argc, char* argv[]) {

    int numInts = 0;
    std::cout << "How many ints: ";
    std::cin >> numInts;

    int* x = NULL;
    if (numInts > 0)
        x = new int[numInts];
    else
    {
        std::cout << "Too few ints!" << std::endl;
        exit(1);
    }

    for (int i=0; i<numInts; i++)
        x[i] = i;

    for (int i=0; i<numInts; i++)
        std::cout << "x[" << i <<"] = " << x[i] << " (" << &x[i] << ")" << std::endl;

    delete [] x;

    return 0;
}
```

When I run this program, I am prompted to enter a number. I entered '5' and got the following output:

```
How many ints: 5
x[0] = 0 (0x10292c5a0)
x[1] = 1 (0x10292c5a4)
x[2] = 2 (0x10292c5a8)
x[3] = 3 (0x10292c5ac)
```

```
x[4] = 4 (0x10292c5b0)
Program ended with exit code: 0
```

The simple program does several things. First, it declares a variable called `numInts` and initializes its value to zero. Second, the program prints out ‘How many ints:’ to the standard console. At this point, nothing happens until the user enters (hopefully) a number. There is no checking that the user actually enters a number. Ideally, we would check that the entry made by the user is valid. In any case, we proceed to declare another variable, this time an `int*` (`int` pointer) variable and initialize its value at the same time we declare it. We initialize its value to be `NULL`. (In C++, we use `NULL` if you want to initialize a pointer variable so that it does not contain an address.) If the variable `numInts` is greater than zero, we go ahead and allocate the desired number of integer variables using the `new` function. If `numInts` is not greater than 0, we print an error and bail out of the program using the `exit` function. Finally, the program initialized the integer variables in the array and prints the value and memory address of each. At the end of the program, we free the memory that was allocated. We do this with the `delete []` function. Every time we dynamically allocate memory using the `new` function, we should remember to free that memory when it is no longer needed using the `delete` function.

For fun, I ran the program again, this time entering ‘1000000’ when prompted to enter how many `int` variables I wanted. The output, minus 990,000 lines, was as follows:

```
How many ints: 1000000
x[0] = 0 (0x103400000)
x[1] = 1 (0x103400004)
x[2] = 2 (0x103400008)
x[3] = 3 (0x10340000c)
x[4] = 4 (0x103400010)
x[5] = 5 (0x103400014)

[many lines not shown]

x[999995] = 999995 (0x1037d08ec)
x[999996] = 999996 (0x1037d08f0)
x[999997] = 999997 (0x1037d08f4)
x[999998] = 999998 (0x1037d08f8)
x[999999] = 999999 (0x1037d08fc)
Program ended with exit code: 0
```


Chapter 5

Conjunction Function

5.1 Making your own functions

Computer programs make extensive use of functions. Functions promote code reuse (why type the same code repeatedly?) and help clarify the logic of the code. So far, we have seen only one function, called `main`. Remember, every C++ program must have a function called `main`. In this chapter, we will explore functions.

Modify your program to read:

Example

```
#include <iostream>

// prototype the new function
int sumTwoInts(int x, int y);

int main(int argc, char* argv[]) {

    int var1 = 3;
    int var2 = 5;
    int sum = sumTwoInts(var1, var2);
    std::cout << "sum = " << sum << std::endl;

    return 0;
}

int sumTwoInts(int x, int y) {
```

```
    return x + y;  
}
```

When run, the output should look like this:

```
sum = 8  
Program ended with exit code: 0
```

This program defines a new function, called `sumTwoInts`. The function is implemented after the `main` function, though it could have been implemented before it (the order doesn't matter). The function `sumTwoInts` is quite simple; it returns the sum of the two `int` variables that are passed to it. The only mysterious portion of this new program is the statement before the `main` function:

```
// prototype the new function  
int sumTwoInts(int x, int y);
```

This statement provides the compiler a heads up that a user-defined function, called `sumTwoInts`, will be used in this file. Besides providing the compiler the name of the function, it informs the compiler that this function will take two `int` variables and return an `int`.

Making new functions is quite easy! Now, let's explore some important details of functions.

5.2 You can pass variables to a function by value or by reference

One of the more confusing aspects of functions is what happens to variables when they are passed into the function. To explore functions, try the following:

Example

```
#include <iostream>  
  
// prototype the new function  
void changeMyVariable(int x);  
  
int main(int argc, char* argv[]) {  
  
    int var = 0;  
    std::cout << "var(before) = " << var << std::endl;
```

```

    changeMyVariable(var);
    std::cout << "var(after) = " << var << std::endl;

    return 0;
}

void changeMyVariable(int x) {

    x = 3;
}

```

Examine this program closely. We define a new function called `changeMyVariable`. This function takes as an argument an `int` variable and returns nothing (hence the `void` as the return statement). The implementation of the `changeMyVariable` function is straight forward. We change the value of `x` to be equal to 3.

The program prints the value of the variable declared in `main` twice: once before and once after the `changeMyVariable` function is called. What do you predict the value of `var` will be after the `changeMyVariable` function is called?

Many new programmers would predict that the value of `var` will be 3 after the `changeMyVariable` function is called. After all, the variable is passed into the function and its value changed there, right? Wrong! Run the program and see for yourself. Your output should look like:

```

var(before) = 0
var(after) = 0
Program ended with exit code: 0

```

Clearly, the value of `var` did not change after the `changeMyVariable` function was called.

The value of `var` does not change after the `changeMyVariable` function is called for the simple reason that the value is copied to a new location when it is passed into the `changeMyVariable` function. This is called passing by value. It is more clear when we print the memory address of `var` in the `main` function and `x` in the `changeMyVariable` function:

Example

```

#include <iostream>

// prototype the new function
void changeMyVariable(int x);

```

```
int main(int argc, char* argv[]) {

    int var = 0;
    std::cout << "var's address: " << &var << std::endl;
    std::cout << "var(before) = " << var << std::endl;
    changeMyVariable(var);
    std::cout << "var(after) = " << var << std::endl;

    return 0;
}

void changeMyVariable(int x) {

    std::cout << "x's address: " << &x << std::endl;
    x = 3;
}
```

When I run the program, I got the following output:

```
var's address: 0x7ffeefbfff67c
var(before) = 0
x's address: 0x7ffeefbfff63c
var(after) = 0
Program ended with exit code: 0
```

You can see from the output that `var` and `x` are different `int` variables residing at different memory addresses. When the function `changeMyVariable` changes the value of `x`, it is not changing the value of `var`. Rather, `x`'s initial value is whatever `var`'s value was. Check this by changing the code to read:

Example

```
#include <iostream>

// prototype the new function
void changeMyVariable(int x);
```

```

int main(int argc, char* argv[]) {

    int var = 11;
    std::cout << "var(before) = " << var << std::endl;
    changeMyVariable(var);
    std::cout << "var(after) = " << var << std::endl;

    return 0;
}

void changeMyVariable(int x) {

    std::cout << "x(before): " << x << std::endl;
    x = 3;
    std::cout << "x(after): " << x << std::endl;
}

```

The output from this modified program clarifies what is going on:

```

var(before) = 11
x(before): 11
x(after): 3
var(after) = 11
Program ended with exit code: 0

```

The variable `var` is initialized to 11. This value is passed to the function `changeMyVariable` where the new variable, `x` within the scope of that function, is initialized to 11. It is then changed to the value 3 and the program exits.

It seems that the `changeMyVariable` function is not acting as we intended. The idea was that we would pass an integer variable into that function and it would change its value to 3. Instead, we get a copy of the variable we want to change and set its value to 3.

How do we change the value of the variable `var` that was instantiated in the `main` function? There are two ways to do this.

First, we could pass in the address of `var` and have the `changeMyVariable` change the value indirectly, by dereferencing the pointer and changing the value at `var`'s address. Let's rewrite the program yet again:

Example

```
#include <iostream>

// prototype the new function
void changeMyVariable(int* x);

int main(int argc, char* argv[]) {

    int var = 0;
    std::cout << "var(before) = " << var << std::endl;
    changeMyVariable(&var);
    std::cout << "var(after) = " << var << std::endl;

    return 0;
}

void changeMyVariable(int* x) {

    (*x) = 3;
}
```

The output for this program looks like:

```
var(before) = 0
var(after) = 3
Program ended with exit code: 0
```

Finally, we have managed to change the value of `var`, that was declared in `main`, in the `changeMyVariable` function. However, we had to do this indirectly. Keep in mind that the variable that is passed into the `changeMyVariable` function is a new variable, in this case an `int*` pointer variable. Its value is initialized when we pass the memory address of `var` to the `changeMyVariable` function when we call it in `main`.

The second way to change the variable is to pass it by reference. We change the program to read:

Example

```
#include <iostream>

// prototype the new function
void changeMyVariable(int& x);

int main(int argc, char* argv[]) {

    int var = 0;
    std::cout << "var's address: " << &var << std::endl;
    std::cout << "var(before) = " << var << std::endl;
    changeMyVariable(var);
    std::cout << "var(after) = " << var << std::endl;

    return 0;
}

void changeMyVariable(int& x) {

    std::cout << "x's address: " << &x << std::endl;
    x = 3;
}
```

The program gives the following output:

```
var's address: 0x7ffeefbff67c
var(before) = 0
x's address: 0x7ffeefbff67c
var(after) = 3
Program ended with exit code: 0
```

Note that `var` (in `main`) and `x` (in `changeMyVariable`) are the same variable. After all, they both are `int` variables at the same memory address. Passing by reference is an alternative method for changing the value of a variable in another function.

5.3 Variables have a scope

Variables have a scope, which is the portion of the code in which they can be accessed, changed, *etc.*. The scope of a variable is defined between `{` and `}`. Rewrite your code as follows:

Example

```
#include <iostream>

int main(int argc, char* argv[]) {

    {
        int x = 0;
        std::cout << "x = " << x << std::endl;
    }

    return 0;
}
```

This program should run and give the output `x = 0`; the additional brackets did not seem to affect the program. Now, let's make a minor change to the program by moving one of the curly brackets:

Example

```
#include <iostream>

int main(int argc, char* argv[]) {

    {
        int x = 0;
    }
    std::cout << "x = " << x << std::endl;

    return 0;
}
```

When I do this, XCode gives me one warning and one error. The error reads, “Use of undeclared identifier ‘x’.” Normally, the scope of `x` would be the entire `main` function (or, at least, the portion of the `main` function after it was declared). However, in this little program, I restricted `x`'s scope to be between another set of curly braces. When I attempted to use `x` outside of its scope, in the `std::cout` statement, the compiler replies with an emphatic “No.”

We could have declared the variable outside of the function:

```
#include <iostream>
```

```
int x; // global x

int main(int argc, char* argv[]) {

    {
        x = 0;
    }
    std::cout << "x = " << x << std::endl;

    return 0;
}
```

Now, the variable `x` is available to any function in that file. It is said to be a ‘global variable.’ Declaring variables to be global in scope is generally not a great idea.

Chapter 6

A Pseudorandom Number Class

6.1 Into the deep end of the pool

In this chapter, we are going to make a random number generator. As it turns out, computers can't generate randomness on their own. All they can do is logical operations and math. So, how do we make randomness from one of the most deterministic tools made by man? The answer is that we don't. Rather, we generate sequences of numbers that in many respects behave randomly. To be more correct, I will modify my original statement: In this chapter, we are going to make a *pseudorandom* number generator.

The sequence of pseudorandom numbers is completely determined by the initial value, called the seed. If we initialize the pseudorandom number generator with the same seed, we will get the same sequence out. If the sequence of random numbers revisits the value for the seed, it will repeat. The period of the pseudorandom number is the length of this repeat. Good pseudorandom number generators will have a very long period. Moreover, the values will be difficult to distinguish from truly random numbers. For example, they shouldn't be correlated with previous values.

You should be wary of using pseudorandom number generators. There is an entire sub-discipline in computer science dedicated to developing better pseudorandom numbers.

We will implement a simple linear congruential generator described by Park and Miller (1988). It is not a state-of-the-art pseudorandom number generator, but it will give you an idea of how apparent randomness can be generated on a computer. It should be good enough for the applications in this book.

A linear congruential generator starts with an initial value of an `int` variable called the seed. The seed value is changed sequentially using the following equation:

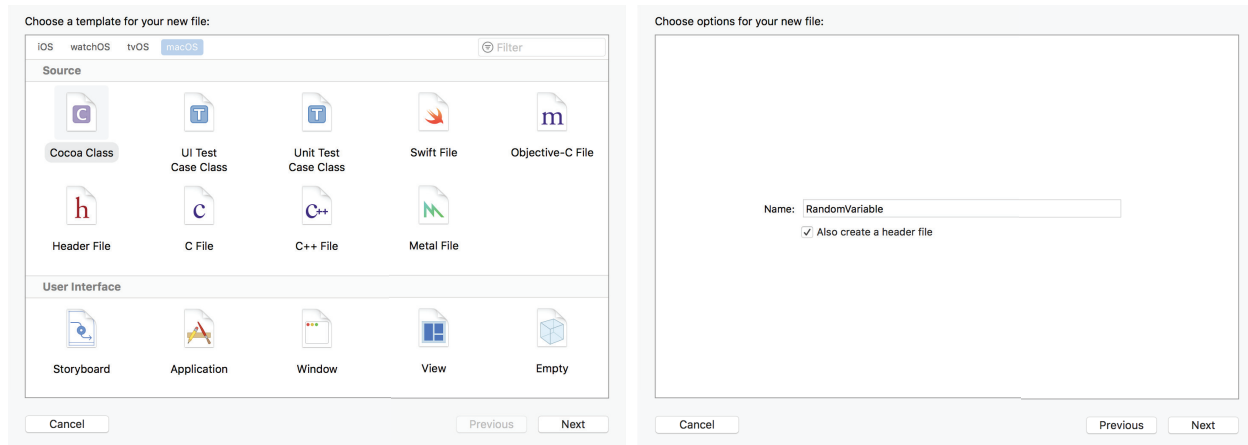
$$s_{i+1} = (a \times s_i + c) \mod m$$

where s_i is the i th value of the seed, a is the multiplier, c is the increment, and m is the modulus. Good pseudorandom number generators that use the linear congruential generator choose a , c , and m wisely. The seed value for a good choice of a , c , and m will bounce between 1 and the maximum value possible in a way that has the appearance of randomness.

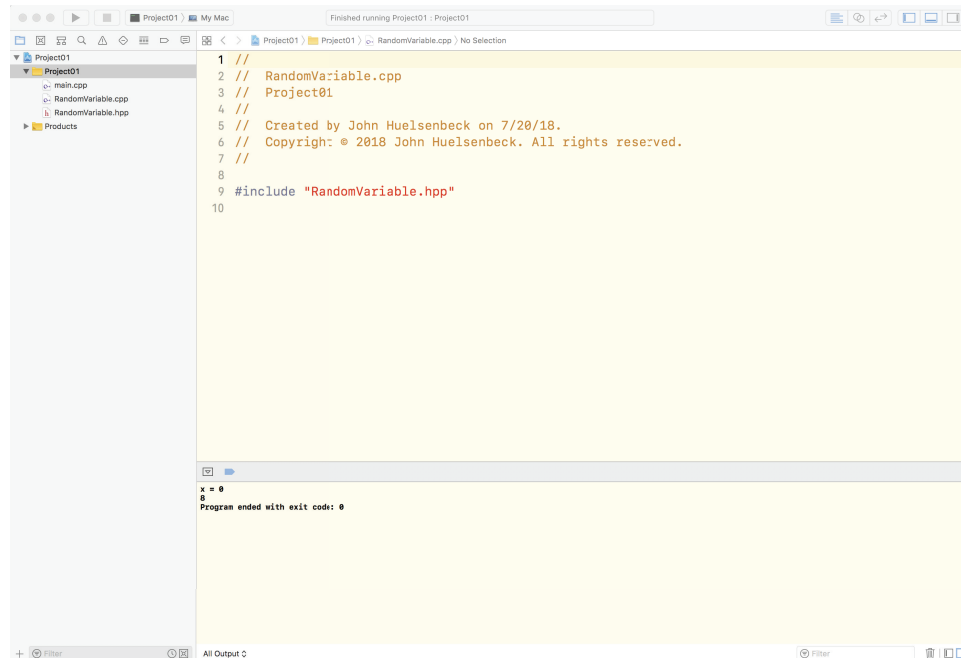
6.2 Making your first class

Make a new project naming it **Project02**. You can follow the instructions in the second chapter of this book to accomplish this. After creating the new project, you should have a single file, **main.cpp**, that contains the **main** function.

Once you have made the new project, you are ready to add a new class. If you are using XCode, select **File > New > File** from the **File** menu item. This will lead to the following windows:



Choose **C++ File** and click on **Next**. In the following window, name the new class **RandomVariable** and click **Next**. This leads to one final window. Simply click **Create** to make your new class files. In the end, your project should look something like this (if you are using XCode):



You created two files, `RandomVariable.hpp` and `RandomVariable.cpp`. You can see them both listed with the `main.cpp` file. When you compile the program, all the files are compiled to form the executable.

Select the file `RandomVariable.hpp`. Add text to that file so it looks like the following:

Example

```
#ifndef RandomVariable_hpp
#define RandomVariable_hpp

class RandomVariable {

    double    uniformRv(void);
    int       seed;
};

#endif
```

Next, select the `RandomVariable.cpp` file. Add text to this file, too:

Example

```
#include "RandomVariable.hpp"

double RandomVariable::uniformRv(void) {

    return 0.0;
}
```

Congratulations! You just made your first class. Eventually, it will be a very cool class, able to generate pseudorandom numbers. We will build to that point, slowly.

What have we done, so far? We defined a class called `RandomVariable`. This class has one `int` variable called `seed` and one function called `uniformRv` associated with it.

6.3 Why do we split the class into .hpp and .cpp files?

One of the more confusing aspects of programming is the fact that a typical program is not implemented in a single file, but rather in many files. As mentioned earlier, the source code for a

program like RevBayes is distributed among several thousand files! Enjoy your good fortune; so far, this program has only three (`main.cpp`, `RandomVariable.hpp`, and `RandomVariable.cpp`).

The class definition was split into two files. Generally speaking, classes are split into two. The `RandomVariable.hpp` file is called the header file. It contains the outline of the class including the variables that are associated with each instance of the class and the functions that are associated with the class. Again, in this class, we only have one variable (`int seed`) and one function `uniformRv`). The class definition is simple:

```
class RandomVariable {

};
```

We include the variables and functions that will be associated with each instance of this class between the curly brackets. You will also note that there are several compiler directives associated with the file:

```
#ifndef RandomVariable_hpp
#define RandomVariable_hpp

#endif
```

The class definition occurs between the compiler directives. The compiler directives ensure that the class is only defined once. Remember, our source code might be distributed among many files. Any file that uses the capabilities of our `RandomVariable` class will include the file at the top, using the include directive:

```
#include "RandomVariable.hpp"
```

Even though we may include the file in dozens (or more) files, we only want the class to be defined once; hence, the compiler directives that guard against this. Alternatively, we could guard against multiple inclusion using the `once` guard:

```
#pragma once

class RandomVariable {

    double  uniformRv(void);
    int      seed;
};
```

Which you use is a matter of personal preference.

If the header file outlines the data and functions associated with each instance of our `RandomVariable` class, the implementation file (`RandomVariable.cpp`) represents the guts of the class. It's in the implementation file that we actually implement the functions. In our implementation file, we only implement the `uniformRv` function:

```
double RandomVariable::uniformRv(void) {

    return 0.0;
}
```

Eventually, this function will generate a pseudorandom number between 0 and 1. For now, it simply returns the value 0. Note that we include the header file that contains the class definition at the head (or top) of this file.

One last thing about the implementation file. You will have noted that the implementation of the `uniformRv` function had `RandomVariable::` before the function name. This is C++'s way of indicating that this function definition is part of the `RandomVariable` class.

Why do we spread the implementation of the class across two files? One idea prevalent in object oriented programming is that the implementation of a class should be hidden from those who use the class. Imagine I wrote a class with functionality that you wanted in your program. You could include the two files I wrote in your project and, voilà, your program now has the capabilities from my class. In order to use that functionality in some file of your program, you would include the header file from my class. The header file of my class should contain all of the information you need to use the functionality of the class. By examining the functions that are available, for example, you would see how to 'interface' with my code. (Hopefully, I would also include useful comments to guide you.) Although the ideal of separating the interface and the implementation is usually not purely implemented in any program that I am aware of, it's a nice idea.

6.4 Instantiating a class

Rewrite the `main.cpp` file to read:

Example

```
#include <iostream>
#include "RandomVariable.hpp"

int main(int argc, char* argv[]) {

    RandomVariable rv;
    double u = rv.uniformRv();
    std::cout << "u(0,1) = " << u << std::endl;

    return 0;
}
```

If you do this, you should come across an error. In XCode, the error reads, “`‘uniformRv’ is a private member of ‘RandomVariable’.`” What is going on? Help!

In C++, variables and functions in a class can be private or public. By default, variables and functions are private. Because we did not indicate otherwise, the compiler assumed that both `seed` and `uniformRv` were both private. To fix the problem, we need to go back to our class definition

in `RandomVariable.hpp` and insert two lines:

Example

```
#ifndef RandomVariable_hpp
#define RandomVariable_hpp

class RandomVariable {

    public:
        double  uniformRv(void);

    protected:
        int      seed;
};

#endif
```

Public variables and functions are available to other parts of the code to use. The private or protected elements of the class, however, are only available to other members of the class. We want other parts of the code to have the ability to get a uniform random variable from this class. Therefore, we make the `uniformRv` function public. On the other hand, the variable `seed` is part of the mechanism that will generate the pseudorandom numbers. We don't want other parts of the program to have the ability to even touch this mechanism. Hence, we hide this variable away. This sort of compartmentalization is a useful feature of C++. In general, you should attempt to keep functions and variables **private** or **protected** unless, of course, it's absolutely necessary to expose the variable or function to the outside world of your code.

With the aside on public versus private/protected completed, let's revisit our `main` function. The first line of that function,

```
RandomVariable rv;
```

declares an instance of the `RandomVariable` class called `rv`. This line of code is analogous to others you have seen, such as `int x`, which makes an instance of an integer variable called `x`. The terminology we use when declaring instances of classes is a bit different than when we declare a variable to be of type `int`. We would say that we are instantiating the `RandomVariable` class or that we made an instance of the `RandomVariable` class. Just like other variables, our new variable `rv` has a scope (the function `main`, in this case). You might have heard that C++ is an 'object-oriented' programming language. Objects are just instances of classes. If it is important to you, you can tell your loved ones that you just made your first object.

If you run the program, the output should look like

```
u(0,1) = 0
Program ended with exit code: 0
```

The code works. Often, I build up portions of the code just as we are doing here: incrementally.

When you make an instance of a class, as we have in our `main` function, several things happened behind the scenes that you are unaware of. First of all, when the variable was created, a function was called that you didn't even know about! This function is called the default constructor function. Again, this function was created even though we never wrote a line of code. Interestingly, we can make our own default constructor that will be called instead. Add to the implementation file, `RandomVariable.cpp`, several new functions, so that the file now looks like:

Example

```
#include <ctime>
#include "RandomVariable.hpp"

RandomVariable::RandomVariable(void) {

    seed = (int)time(NULL);
}

RandomVariable::RandomVariable(int x) {

    seed = x;
}

double RandomVariable::uniformRv(void) {

    return 0.0;
}

double RandomVariable::uniformRv(double lower, double upper) {

    return 0.0;
}
```

Modify the header file to reflect the changes you made in the implementation file:

Example

```
#ifndef RandomVariable_hpp
```

```

#define RandomVariable_hpp

class RandomVariable {

    public:
        RandomVariable(void);
        RandomVariable(int x);
        double uniformRv(void);
        double uniformRv(double lower, double upper);

    protected:
        int seed;
};

#endif

```

You should test that you did everything correctly by attempting to run the program.

The default constructor has the same name as the class and does not return a variable (in fact, it doesn't even have the keyword `void` to indicate this, like normal functions that don't return a value). If we wanted to replicate, exactly, the default constructor that was created by the compiler, we would simply include the following code:

```

RandomVariable::RandomVariable(void) {

}

```

Our implementation of the default constructor, however,

```

RandomVariable::RandomVariable(void) {

    seed = (int)time(NULL);
}

```

sets the `seed` variable to be equal to the current time, obtained using the `time` function. The variable that is returned by the `time` function is not an `int` variable, so we cannot directly equate it to `seed`. The `time` function returns a variable of type `time_t`. However, we can force the `time_t` variable to act as an `int` using the casting argument, which was the `(int)` before the `time(NULL)`. Casting can be dangerous. You can't always sensibly cast one variable type into another. In this case, however, you can sensibly cast a `time_t` variable into an `int` variable. All is well.

Add a line of code to the default constructor, so that it now reads:

Example

```
RandomVariable::RandomVariable(void) {  
  
    seed = (int)time(NULL);  
    std::cout << "Default constructor. The seed equals " << seed << std::endl;  
}
```

In order to get the code to compile, you will need to add `iostream` using the `include` compiler directive to the `RandomVariable.cpp` file. When I ran the program, I obtained the following output:

```
Default constructor. The seed equals 1532124958  
u(0,1) = 0  
Program ended with exit code: 0
```

The important point is this: We never explicitly called the default constructor. It was called for us when `rv` was created.

Constructors are obvious places to initialize variables. For our `RandomVariable` class, it makes sense to initialize the `seed` variable when an instance of the class is created. Moreover, because we want a different sequence of pseudorandom numbers every time we run the program, we should initialize the `seed` variable to something that changes, like time. Many implementations of pseudorandom number generators use the computer's internal clock to initialize the seed.

What if we want to initialize the `seed` variable to some other value? We have given our program the ability to do just this. In the code, above, we implemented a second constructor! This one,

```
RandomVariable::RandomVariable(int x) {  
  
    seed = x;  
}
```

Takes an `int` argument and sets the `seed` variable to be equal to it. Add a line to this second constructor, so it reads:

Example

```
RandomVariable::RandomVariable(int x) {  
  
    seed = x;  
    std::cout << "Alternate constructor. The seed equals " << seed << std::endl;  
}
```

Now, go to the main function and rewrite it so it reads:

Example

```
#include <iostream>
#include "RandomVariable.hpp"

int main(int argc, char* argv[]) {

    RandomVariable rv1;
    RandomVariable rv2(3);
    RandomVariable rv3(1929);
    RandomVariable rv4;

    return 0;
}
```

When I ran the program, I got the following output:

```
Default constructor. The seed equals 1532125424
Alternate constructor. The seed equals 3
Alternate constructor. The seed equals 1929
Default constructor. The seed equals 1532125424
Program ended with exit code: 0
```

We made four different instances of the `RandomVariable` class. This means that we have four instances of seed that were created, one for each. `rv1` and `rv4` were created using the default constructor which sets the seed using the current time. Both of those objects have the same value for the `seed` variable. They will produce the same sequences of pseudorandom numbers. The computer created the four instances of `RandomVariable` so quickly that the time didn't have a chance to turn over, even by one! The other two instances of `RandomVariable`, `rv2` and `rv3`, were created using the alternative constructor that sets the seed to some value that is passed in.

One nice feature of C++ is function overloading. Different functions can have the same name, as long as the arguments that are passed to the functions are different. In our `RandomVariable` class, we see function overloading in two places: for the constructors

```
RandomVariable(void);
RandomVariable(int x);
```

and in the functions `uniformRv`

```
double uniformRv(void);
double uniformRv(double lower, double upper);
```

Even though the functions have the same name, the compiler can tell them apart because the arguments that are passed to the functions are different. The compiler can tell which function is being called, implicitly, by examining the list of arguments that are passed to it. We were able to indicate to the compiler which constructor we wanted to be called when the objects were instantiated: `RandomVariable rv` calls the default constructor, which doesn't take any arguments, whereas `RandomVariable rv(3)` calls the alternative constructor which takes a single `int` as an argument.

6.5 Implementing the `uniformRv` function

We will implement the pseudorandom number generator described by Park and Miller (1988). Change the code for the `uniformRv(void)` function in the `RandomVariable.cpp` file to read:

Example

```
double RandomVariable::uniformRv(void) {  
  
    int hi = seed / 127773;  
    int lo = seed % 127773;  
    int test = 16807 * lo - 2836 * hi;  
    if (test > 0)  
        seed = test;  
    else  
        seed = test + 2147483647;  
    return (double)(seed) / (double)2147483647;  
}
```

The code for pseudorandom number generators is always disturbing to look at. The non-randomness of a pseudorandom number generator sort of smacks you in the face when you see the code. You should be feeling a little queasy at this point.

Clearly, the `seed` variable is changed every time the `uniformRv` function is called. The number that is returned is the new value for the seed divided by its maximum possible value. We use casting, again, to force the compiler to treat `seed` as a `double` (real-valued number) when the division occurs in the last line. If we didn't do this, the function would return 0 every time.

We have implemented the `uniformRv(void)` function, but have the `uniformRv(double lower, double upper)` remaining to be implemented. Go to that function, and add the following code:

Example

```
double RandomVariable::uniformRv(double lower, double upper) {  
  
    return ( lower + uniformRv() * (upper - lower) );  
}
```

This function uses the function that returns a uniformly-distributed random number on the interval $(0, 1)$ to generate a random number uniformly-distributed on the interval $(Lower, Upper)$.

Now for some fun! Let's test our random number generator. Modify your main function so it reads:

Example

```
#include <iostream>  
#include "RandomVariable.hpp"  
  
int main(int argc, char* argv[]) {  
  
    RandomVariable rv;  
    for (int i=0; i<100; i++)  
    {  
        double u = rv.uniformRv();  
        std::cout << i << " -- " << u << std::endl;  
    }  
  
    return 0;  
}
```

When you run this program, the output should be 100 pseudorandom numbers, uniformly distributed on the interval $(0, 1)$.

Modify the program so that it starts with the same seed every time. Confirm that the sequence of random numbers is the same every time the program is run.

6.6 Implementing the exponentialRv function

Besides being a nifty example of function overloading, our function `uniformRv(double lower, double upper)` demonstrates a commonly-used method for generating random numbers that are not `uniform(0,1)` random variables. That function transforms a `uniform(0,1)` random number into a `uniform(lower,upper)` random number. We can use the same idea to generate different types of random variables. Here, we will make one more random variable before moving on from our discussion of classes.

The exponential probability distribution accurately models the waiting time until an event occurs when that event occurs at a constant rate, λ . The exponential probability distribution is a continuous distribution with density function, $f(t) = \lambda e^{-\lambda t}$, for $t > 0, \lambda > 0$. How can we generate an exponentially-distributed random number?

The idea is to transform the `uniform(0,1)` random variable into an `exponential(λ)` random variable. We know that if we integrate over all possible values of t , that the overall probability is one (*i.e.*, the exponential is a probability distribution):

$$\int_0^{\infty} \lambda e^{-\lambda x} dx = 1$$

The key to transforming our uniform into an exponential is to generate a `uniform(0,1)` random variable called u and set the integral equal to that value:

$$\int_0^t \lambda e^{-\lambda x} dx = u$$

We can solve for t as follows:

$$\begin{aligned} \int_0^t \lambda e^{-\lambda x} dx &= u \\ -e^{-\lambda x} \Big|_0^t &= u \\ -e^{-\lambda t} - -e^0 &= u \\ 1 - e^{-\lambda t} &= u \\ e^{-\lambda t} &= 1 - u \\ -\lambda t &= \ln(1 - u) \\ t &= -\frac{1}{\lambda} \ln(1 - u) \end{aligned}$$

Here, t is an `exponential(λ)` random variable. It was obtained by transforming our `uniform(0,1)` random variable, u . Equivalently, we could use the equation $t = -\frac{1}{\lambda} \ln(u)$.

Now that we have the math out of the way, let's implement our exponential random number generator. Add to the `RandomVariable.cpp` file the following function:

Example

```
double RandomVariable::exponentialRv(double lambda) {  
  
    return -log(uniformRv()) / lambda;  
}
```

You will have to do two things to get this to compile. First, include the function profile in the header file for the class. Second, you will need to include the `cmath` header file in the implementation file. (The natural log function, `log`, is implemented in `cmath`.)

Once you have successfully implemented the `exponentialRv` function, play around with it. Generate a bunch of exponentially-distributed random numbers. What is the mean of the numbers you generated? They should be near to the expectation for the exponential, $E(X) = 1/\lambda$. Confirm this.

Chapter 7

Markov chain Monte Carlo

7.1 Some theory

Markov chain Monte Carlo (MCMC) is a numerical method for approximating high-dimensional integrals and/or summations. The method was first described by Metropolis et al. (1953) and later modified by (Hastings, 1970)¹. The algorithm that we will implement in this chapter is often referred to as the Metropolis-Hastings algorithm.

Use of the Metropolis-Hastings algorithm was mostly restricted to the field of statistical physics until it was discovered by statisticians in the mid 1980s (Geman and Geman, 1984). Statisticians, as a group, were largely unaware of the numerical method until Gelfand and Smith (1990) provided a description of the method as a way to approximate intractable probability densities. It is fair to say that MCMC has transformed the field of statistics, and in particular, Bayesian statistics.

Bayesian statisticians are interested in the posterior probability distribution of a parameter, θ , which can be calculated using Bayes's theorem as

$$\mathbb{P}(\theta|D) = \frac{\mathbb{P}(D|\theta) \mathbb{P}(\theta)}{\mathbb{P}(D)}$$

Here, $\mathbb{P}(\theta|D)$ is the posterior probability distribution of the parameter. The posterior probability is a conditional probability: the probability of the parameter conditioned on the observations, D . The other factors in the equation include the likelihood $[\mathbb{P}(D|\theta)]$, prior probability distribution of the parameter $[\mathbb{P}(\theta)]$, and marginal likelihood $[\mathbb{P}(D)]$.

How does Bayesian analysis work in practice? Consider an experiment in which a coin is repeatedly tossed with the objective to estimate the probability that heads appears on a single toss of the coin, a parameter we call θ . We observe x heads on n tosses of the coin. In a Bayesian analysis, the objective is to calculate the posterior probability of the parameter, which for coin tossing is

$$f(\theta|x) = \frac{f(x|\theta)f(\theta)}{\int_0^1 f(x|\theta)f(\theta) d\theta}$$

¹Look at the full list of authors for the Metropolis et al. (1953) paper. If you know your U.S. history, you will recognize E. Teller as the father of the American hydrogen bomb.

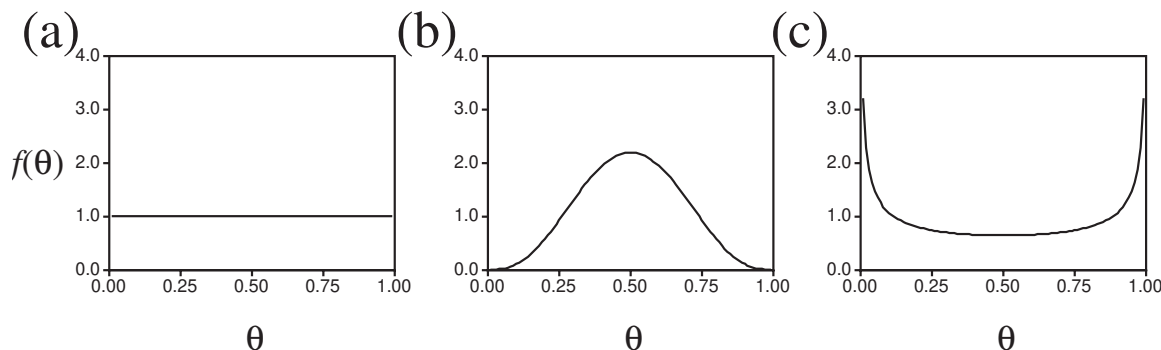


Figure 7.1: The Beta distribution can take a variety of shapes depending on the values of the parameters α and β . Here, (a) $\alpha = \beta = 1$, (b) $\alpha = \beta = 4$, and (c) $\alpha = \beta = 1/2$.

The likelihood, $f(x | \theta)$, is given by the binomial probability distribution,

$$f(x | \theta) = \binom{n}{x} \theta^x (1 - \theta)^{n-x}$$

where the binomial coefficient is $\binom{n}{x} = \frac{n!}{x!(n-x)!}$. In addition to the likelihood function, however, we must also specify a prior probability distribution for the parameter, $f(\theta)$. This prior distribution should describe the investigator's beliefs about the hypothesis before the experiment was performed. The problem, of course, is that different people may have different prior beliefs. In this case, it makes sense to use a prior distribution that is flexible, allowing different people to specify different prior probability distributions and also allowing for an easy investigation of the sensitivity of the results to the prior assumptions. A Beta distribution is often used as a prior probability distribution for the binomial parameter. The Beta distribution has two parameters, α and β . Depending upon the specific values chosen for α and β , one can generate a large number of different prior probability distributions for the parameter θ . Figure 7.1 shows several possible prior distributions for coin tossing. Figure 7.1a shows a uniform prior distribution for the probability of heads appearing on a single toss of a coin. In effect, a person who adopts this prior distribution is claiming total ignorance of the dynamics of coin tossing. Figure 7.1b shows a prior distribution for a person who has some experience tossing coins; anyone who has tossed a coin realizes that it is impossible to predict which side will face up when tossed, but that heads appears about as frequently as tails, suggesting more prior weight on values around $\theta = 0.5$ than on values near $\theta = 0$ or $\theta = 1$. Lastly, Figure 7.1c shows a prior distribution for a person who suspects he is being tricked. Perhaps the coin that is being tossed is from a friend with a long history of practical jokes, or perhaps this friend has tricked the investigator with a two-headed coin in the past. Figure 7.1c, then, might represent the 'trick-coin' prior distribution.

Besides being flexible, the Beta prior probability distribution has one other admirable property: when combined with a binomial likelihood, the posterior distribution also has a Beta probability distribution (but with the parameters changed). Prior distributions that have this property — that is, the posterior probability distribution has the same functional form as the prior distribution — are called conjugate priors in the Bayesian literature. The Bayesian treatment of the coin tossing

experiment can be summarized as follows:

Prior [$f(\theta)$, Beta]	Likelihood [$f(x \theta)$, Binomial]	Posterior [$f(\theta x)$, Beta]
$\frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}\theta^{\alpha-1}(1-\theta)^{\beta-1}$	$\binom{n}{x}\theta^x(1-\theta)^{n-x}$	$\frac{\Gamma(\alpha+\beta+n)}{\Gamma(\alpha+x)\Gamma(\beta+n-x)}\theta^{\alpha+x-1}(1-\theta)^{\beta+n-x-1}$

[The gamma function, not to be confused with the gamma probability distribution, is defined as $\Gamma(y) = \int_0^\infty u^{y-1}e^{-u} du$. $\Gamma(n) = (n-1)!$ for integer $n = 1, 2, 3, \dots$ and $\Gamma(\frac{1}{2}) = \pi$.] We started with a Beta prior distribution with parameters α and β . We used a binomial likelihood, and after some use of our undergraduate calculus we calculated the posterior probability distribution, which is also a Beta distribution but with parameters $\alpha + x$ and $\beta + n - x$.

Figure 7.2 shows the relationship between the prior probability distribution, likelihood, and posterior probability distribution of θ for two different cases, differing only in the number of coin tosses. The posterior probability of a parameter is a compromise between the prior probability and the likelihood. When the number of observations is small, as is the case for Figure 7.2a, the posterior probability distribution is similar to the prior distribution. However, when the number of observations is large, as is the case for Figure 7.2b, the posterior probability distribution is dominated by the likelihood.

Above, I casually claimed that the posterior distribution was a Beta distribution. Working out the math for this problem is informative. The posterior probability distribution for the coin tossing problem can be calculated analytically. Here, we assume a binomial probability distribution for the likelihood, and a Beta prior distribution on θ , which means the posterior probability distribution

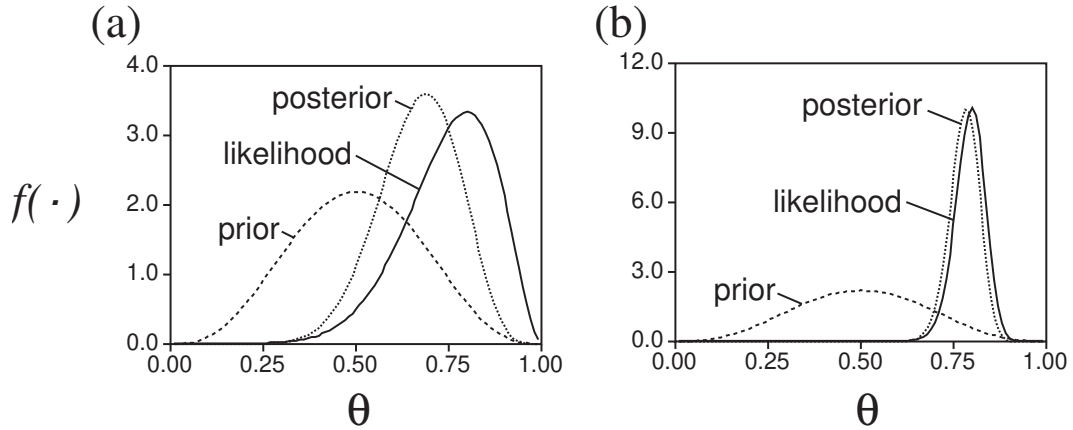


Figure 7.2: As more data are collected for the coin-tossing example, the investigator's prior opinions play a smaller role in the conclusions. (a) The prior distribution, likelihood function, and posterior probability density when $\alpha = \beta = 4$, $n = 10$ and $x = 8$. (b) The prior distribution, likelihood function, and posterior probability density when $\alpha = \beta = 4$, $n = 100$ and $x = 80$

of the parameter θ is:

$$f(\theta|x) = \frac{\binom{n}{x} \theta^x (1-\theta)^{n-x} \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} \theta^{\alpha-1} (1-\theta)^{\beta-1}}{\int_0^1 \binom{n}{x} \theta^x (1-\theta)^{n-x} \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} \theta^{\alpha-1} (1-\theta)^{\beta-1} d\theta}$$

Note that the denominator involves integration over all possible values of the parameter θ . Specifically, the probability of heads on a single toss of a coin, θ , can take values between 0 and 1.

Evaluating the integral in the denominator depends upon the following observation: The Beta probability distribution, like all continuous probability distributions, evaluates to one when integrated over all possible values for the parameter:

$$\begin{aligned} \int_0^1 \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1} (1-x)^{\beta-1} dx &= 1 \\ \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} \int_0^1 x^{\alpha-1} (1-x)^{\beta-1} dx &= 1 \\ \int_0^1 x^{\alpha-1} (1-x)^{\beta-1} dx &= \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)} \end{aligned}$$

The denominator in the equation for the posterior probability distribution of θ , above, is

$$\begin{aligned} &\int_0^1 \binom{n}{x} \theta^x (1-\theta)^{n-x} \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} \theta^{\alpha-1} (1-\theta)^{\beta-1} d\theta \\ &\frac{\binom{n}{x} \Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} \int_0^1 \theta^x (1-\theta)^{n-x} \theta^{\alpha-1} (1-\theta)^{\beta-1} d\theta \\ &\frac{\binom{n}{x} \Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} \int_0^1 \theta^{\alpha+x-1} (1-\theta)^{\beta+n-x-1} d\theta \end{aligned}$$

Because $\int_0^1 x^{\alpha-1} (1-x)^{\beta-1} dx = \Gamma(\alpha)\Gamma(\beta)/\Gamma(\alpha+\beta)$, this means that $\int_0^1 x^{\alpha+x-1} (1-x)^{\beta+n-x-1} dx = \Gamma(\alpha+x)\Gamma(\beta+n-x)/\Gamma(\alpha+\beta+n)$. The posterior probability distribution for θ , then, is

$$\begin{aligned} f(\theta|x) &= \frac{\binom{n}{x} \theta^x (1-\theta)^{n-x} \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} \theta^{\alpha-1} (1-\theta)^{\beta-1}}{\frac{\binom{n}{x} \Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} \frac{\Gamma(\alpha+x)\Gamma(\beta+n-x)}{\Gamma(\alpha+\beta+n)}} \\ &= \frac{\Gamma(\alpha+\beta+n) \theta^x (1-\theta)^{n-x} \theta^{\alpha-1} (1-\theta)^{\beta-1}}{\Gamma(\alpha+x)\Gamma(\beta+n-x)} \\ &= \frac{\Gamma(\alpha+\beta+n) \theta^{\alpha+x-1} (1-\theta)^{\beta+n-x-1}}{\Gamma(\alpha+x)\Gamma(\beta+n-x)} \end{aligned}$$

which is a Beta probability distribution with parameters $\alpha+x$ and $\beta+n-x$.

I will make one comment about the analytical solution of the posterior probability distribution for the coin-tossing problem: **it involved hard math!** Certainly, working through the math was at the edge of my ability. However, for problems that are only slightly more complicated than coin tossing (*i.e.*, almost all other statistical problems), it is not possible to obtain the posterior probability distribution analytically. Until the advent of computers and algorithms for approximating high-dimensional integrals/summation, people were up a river without a paddle if their problem was somewhat complicated (*i.e.*, problems that are actually interesting).

7.2 MCMC

Even though we know the posterior probability distribution for the coin-tossing problem, we will act as if we don't and approximate the posterior probability distribution using MCMC. The goal of MCMC is to construct a Markov chain that has as its state space the parameters of the model and a stationary distribution that is the posterior probability of the parameters. How do we construct a Markov chain that has these properties? The answer: Use the Metropolis-Hastings algorithm. The steps of the Metropolis-Hastings algorithm are as follows:

1. Call the current state of the Markov chain θ . If this is the first cycle of the chain, then initialize θ to some value. (Perhaps initialize θ by choosing a value from the prior distribution.)
2. Propose a new value for θ , called θ' . The details of the proposal mechanism are at the discretion of the programmer. However, he/she must follow some rules. For one, the proposal mechanism that is coded must involve random numbers. That is, the proposal mechanism cannot be a deterministic one. Some other requirements are that every state must be potentially reachable given enough applications of the proposal mechanism and the proposal mechanism cannot result in a periodic chain. (The last requirement is generally not an issue.) Finally, you should be able to calculate the probability of proposing the proposed state, $q(\theta \rightarrow \theta')$ as well as the imagined reverse move, which is not actually made in computer memory, $q(\theta' \rightarrow \theta)$.
3. Calculate the probability of accepting the proposed state as the next state of the Markov chain. This probability is the ratio of the posterior probabilities of the proposed and current states times the Hastings ratio:

$$R = \min \left(1, \frac{f(\theta'|X)}{f(\theta|X)} \times \underbrace{\frac{q(\theta' \rightarrow \theta)}{q(\theta \rightarrow \theta')}}_{\text{Hastings Ratio}} \right)$$

It does not appear that we can solve this. After all, the entire point of MCMC is to approximate probability distributions that we cannot solve analytically. Here, things seem especially intractable, because we need to calculate the ratio of two probability distributions that we cannot calculate analytically! Relax. Let's rewrite the ratio of the posterior probabilities using Bayes's theorem:

$$R = \min \left(1, \frac{f(X|\theta')f(\theta')/f(X)}{f(X|\theta)f(\theta)/f(X)} \times \frac{q(\theta' \rightarrow \theta)}{q(\theta \rightarrow \theta')} \right)$$

Note that the marginal likelihood, $f(X)$, cancels. Importantly, it was the marginal likelihood that we cannot calculate analytically. The portions that are left over, after cancelling $f(X)$, are readily calculable. In the end, our acceptance probability is:

$$R = \min \left(1, \underbrace{\frac{f(X|\theta')}{f(X|\theta)}}_{\text{Likelihood Ratio}} \times \underbrace{\frac{f(\theta')}{f(\theta)}}_{\text{Prior Ratio}} \times \underbrace{\frac{q(\theta' \rightarrow \theta)}{q(\theta \rightarrow \theta')}}_{\text{Hastings Ratio}} \right)$$

4. Generate a uniform(0,1) random variable called u . If $u < R$, then accept the proposed state as the next state of the Markov chain, setting $\theta = \theta'$. Otherwise, the chain remains in its current state.
5. Go to Step # 2.

The steps of the Metropolis-Hastings algorithm are repeated many thousands, or millions, of times. The states that are visited form a Markov chain. The key to MCMC is that the sampled states are valid, albeit dependent, draws from the posterior probability distribution.

7.3 Coding MCMC for coin tossing

Make a new Project in your IDE. We will need to generate pseudorandom numbers for our MCMC algorithm, so include in the project the `RandomVariable` class that you made in the last chapter.

We will assume that you have tossed a coin $n = 100$ times and observed $x = 43$ heads. The likelihood function will be the binomial probability distribution. We will assume a flat prior in which all values of the parameter θ , which is interpreted as the probability of heads on a single toss of the coin, have equal probability. This is equivalent to a Beta prior probability distribution with parameters $\alpha = \beta = 1$.

We will build this program very slowly. First, let's confirm that we can generate some random numbers in the new project. Rewrite your `main` function so that it reads:

Example

```
#include <iostream>
#include "RandomVariable.hpp"

int main(int argc, const char * argv[]) {

    // instantiate an object for generating random numbers
    RandomVariable rv;

    // test the random number generator
    for (int i=0; i<100; i++)
    {
        double u = rv.uniformRv();
        std::cout << i << " -- " << u << std::endl;
    }

    return 0;
}
```

Confirm that the program compiles and that the output is 100 pseudorandom numbers. Many programmers work in this way. They build up the program incrementally, testing it at each step. We should have a compiling program that can generate random numbers. We are ready to continue.

Next, we will add some code that acts as the world's most miserable user-interface. Modify your `main` function to read:

Example

```
int main(int argc, const char * argv[]) {

    // instantiate an object for generating random numbers
    RandomVariable rv;

    // this user-interface sucks!
    int numTosses      = 100;
    int numHeads       = 43;
    int numTails       = numTosses - numHeads;
    int chainLength    = 1000;
    int printFrequency = 10;
    int sampleFrequency = 1;
    double window      = 0.1;

    return 0;
}
```

All we added were some variables that are going to be necessary for our MCMC analysis of the coin-tossing problem. Why is this a miserable user-interface? Essentially, anybody who wanted to use our program and had results that were different than $x = 43$ heads on $n = 100$ tosses of a coin would have to go into the source code, change the numbers to reflect their results, and recompile the code. Imagine if programs such as MrBayes instructed the user to modify the source code such that a particular string, buried deep in the code, reflected the path to the file that contained the data to be analyzed. The authors of the code would be hunted down and beaten up by their users.

Now that we have our user interface, we will write the main body of the MCMC loop:

Example

```
int main(int argc, const char * argv[]) {
```

```
// instantiate an object for generating random numbers
RandomVariable rv;

// this user-interface sucks!
int numTosses      = 100;
int numHeads       = 43;
int numTails       = numTosses - numHeads;
int chainLength    = 1000;
int printFrequency = 10;
int sampleFrequency = 1;
double window      = 0.1;

// initialize theta to some value

// run the Markov chain
for (int n=1; n<=chainLength; n++)
{
    // propose a new value for theta

    // calculate the probability of accepting thetaPrime as
    // the next state of the chain

    // accept or reject thetaPrime, and update the state of the chain
}

return 0;
}
```

Again, we have made an incremental change. All we added was a loop with comments indicating what we need to do at different portions of the code to accomplish the steps of the Metropolis-Hastings algorithm. Again, confirm that the program compiles and runs. (There should be no output, just an indication that the program successfully exited.)

The first step is to declare a variable that will hold the current state of the chain, and to initialize this variable to some value. Add to the code, at the appropriate place, the following code:

Example

```
// initialize theta to some value
double theta = rv.uniformRv();
```

Here, we declare a variable named `theta` to be of type `double`. This variable can hold a real-valued number. We initialize it using the pseudorandom number generator. Initializing `theta` by drawing from a `uniform(0,1)` probability distribution is equivalent to drawing from our flat prior distribution.

Now that we have initialized `theta`, we move into the MCMC loop (the `for` loop). First, we have to propose a new value for `theta`. Our proposal mechanism should adhere to the rules we outlined in Step # 2 of our Metropolis-Hastings algorithm description. I would suggest that we use a sliding window mechanism for proposing a new value for `theta`. The idea is to center a window of fixed width on the current value of `theta`. Our new value for `theta` is chosen randomly, with uniform probability, from the window. This means that our next value for the parameter will be similar to the current value. Modify the appropriate part of the code in the `main` function to read:

Example

```
// propose a new value for theta
double thetaPrime = theta + (rv.uniformRv() - 0.5) * window;
if (thetaPrime < 0.0)
    thetaPrime = -thetaPrime;
else if (thetaPrime > 1.0)
    thetaPrime = 2.0 - thetaPrime;
```

Convince yourself that this proposal mechanism will propose a new value for `theta`, here called `thetaPrime`, that is up to `window/2` larger or `window/2` smaller than `theta`. The proposal probability for the forward move is $q(\text{theta} \rightarrow \text{thetaPrime}) = 1/\text{window}$. The probability of the reverse move, which we don't actually realize in computer memory, is $q(\text{thetaPrime} \rightarrow \text{theta}) = 1/\text{window}$. Also, note that after we propose the new value `thetaPrime`, we check that it is a valid value. The probability of heads for a coin must be between 0 and 1. However, it's possible for our sliding window mechanism to propose values outside of that range. In this implementation, we reflect the value back into the valid region. Another possible solution to the problem is to simply reject values that are outside the valid range.

Next, we need to calculate the probability of accepting the proposed state. As pointed out in our description of the Metropolis-Hastings algorithm, the acceptance probability is the product of

the likelihood ratio, prior ratio, and Hastings (or proposal) ratio:

$$R = \min \left(1, \frac{\binom{n}{x} \theta'^x (1 - \theta')^{n-x}}{\binom{n}{x} \theta^x (1 - \theta)^{n-x}} \times \frac{1}{1} \times \frac{1/w}{1/w} \right)$$

where n is `numTosses`, x is `numHeads`, and w is `window` in the C++ code. Also, note that the binomial coefficients, $\binom{n}{x}$, cancel, so we won't be dealing with them in the code. Finally, we will be working with the natural log of the probabilities, and not the probabilities directly. We do this to avoid underflow in the probabilities.

Modify the appropriate part of the code in the `main` function with the following code snippet:

Example

```
// calculate the probability of accepting thetaPrime as
// the next state of the chain
double lnLikelihoodRatio = (numHeads*log(thetaPrime)+numTails*log(1.0-thetaPrime)) -
                           (numHeads*log(theta      )+numTails*log(1.0-theta      ));
double lnPriorRatio = 0.0;    // natural log of 1.0
double lnHastingsRatio = 0.0; // natural log of 1.0
double lnR = lnLikelihoodRatio + lnPriorRatio + lnHastingsRatio;
double R = 0.0;
if (lnR < -300.0)
    R = 0.0;
else if (lnR > 0.0)
    R = 1.0;
else
    R = exp(lnR);
```

Because you are now using the `log` and `exp` functions, you will need to include the `cmath` file in this file. This code breaks the calculation into three parts. The first part calculates the natural log of the likelihood ratio, a variable called `lnLikelihoodRatio` in the code. Remember that taking the log turns multiplication into addition and division into subtraction. The next two quantities — `lnPriorRatio` and `lnHastingsRatio` — are the logs of the prior and Hastings ratios, both of which are one. They do not need to be included in the code because they are both zero ($\ln(1) = 0$). However, I include them both for pedagogical reasons, to remind you that in our case both ratios just happen to be one. If we had used a different prior or proposal mechanism, this wouldn't necessarily be true.

The log of the acceptance probability is simply the sum of the log of the likelihood, prior, and proposal ratios. This quantity is called `lnR` in the code. The last part of the code exponentiates the `lnR` safely. We want a probability, not a log probability. We set the acceptance probability, `R`, to be equal to zero if `lnR` is less than -300.0. We do this because if we were to encounter an `lnR` less than -300.0, we would likely encounter an underflow error if we attempted exponentiation.

Finally, we need to accept or reject the proposed state of the Markov chain. Add the following fragment of code to the `main` function:

Example

```
// accept or reject thetaPrime, and update the state of the chain
double u = rv.uniformRv();
if (u < R)
    theta = thetaPrime;
```

Congratulations! You have just written your first MCMC program. Your complete `main` function should look like:

Example

```
#include <cmath>
#include <iostream>
#include "RandomVariable.hpp"

int main(int argc, const char * argv[]) {

    // instantiate an object for generating random numbers
    RandomVariable rv;

    // this user-interface sucks!
    int numTosses      = 100;
    int numHeads       = 43;
    int numTails       = numTosses - numHeads;
    int chainLength    = 1000;
    int printFrequency = 10;
    int sampleFrequency = 1;
    double window       = 0.1;

    // initialize theta to some value
    double theta = rv.uniformRv();
```

```

// run the Markov chain
for (int n=1; n<=chainLength; n++)
{
    // propose a new value for theta
    double thetaPrime = theta + (rv.uniformRv() - 0.5) * window;
    if (thetaPrime < 0.0)
        thetaPrime = -thetaPrime;
    else if (thetaPrime > 1.0)
        thetaPrime = 2.0 - thetaPrime;

    // calculate the probability of accepting thetaPrime as
    // the next state of the chain
    double lnLikelihoodRatio = (numHeads*log(thetaPrime)+numTails*log(1.0-thetaPrime)) -
                                (numHeads*log(theta) + numTails*log(1.0-theta));
    double lnPriorRatio = 0.0; // natural log of 1.0
    double lnHastingsRatio = 0.0; // natural log of 1.0
    double lnR = lnLikelihoodRatio + lnPriorRatio + lnHastingsRatio;
    double R = 0.0;
    if (lnR < -300.0)
        R = 0.0;
    else if (lnR > 0.0)
        R = 1.0;
    else
        R = exp(lnR);

    // accept or reject thetaPrime, and update the state of the chain
    double u = rv.uniformRv();
    if (u < R)
        theta = thetaPrime;
}

return 0;
}

```

Go ahead and run the program. A bit underwhelming, isn't it? The reason, of course, is because we don't see any output. The analysis runs, presumably does exactly what we want, then the program exits.

Ideally, we would be able to witness the MCMC analysis as it proceeds. We can do this by adding code that prints the current state of the Markov chain to the standard console.

Add code before and after the code segment that accepts or rejects the proposed state, such

that a portion of the code looks like:

Example

```
// print (part 1)
if (n % printFrequency == 0)
{
    std::cout << std::setw(5) << n << " -- ";
    std::cout << std::fixed << std::setprecision(3) << theta << " -> ";
    std::cout << std::fixed << std::setprecision(3) << thetaPrime << " ";
}

// accept or reject thetaPrime, and update the state of the chain
double u = rv.uniformRv();
bool isAccepted = false;
if (u < R)
{
    theta = thetaPrime;
    isAccepted = true;
}

// print (part 2)
if (n % printFrequency == 0)
{
    if (isAccepted == true)
        std::cout << "(Accepted)";
    else
        std::cout << "(Rejected)";
    std::cout << std::endl;
}
```

Because we are using formatting commands, you will need to add the `iomanip` file to the file, using the `#include <iomanip>` compiler directive. The statement `if (n % printFrequency == 0)` divides the current MCMC cycle, `n` by `printFrequency`. If the remainder is 0, then we go ahead and execute the portion of the code in the curly brackets after the `if` statement. This ensures that we print the state of the Markov chain to the console ever `printFrequency` MCMC cycles. Printing to the console takes place in two parts: both before and after we have decided to accept or reject the proposed state. This code does a nifty job of printing the state of the chain to the screen. The last 10 lines of my output look like:

```
910 -- 0.483 -> 0.440 (Accepted)
```



```

920 -- 0.429 -> 0.451 (Accepted)
930 -- 0.324 -> 0.344 (Accepted)
940 -- 0.430 -> 0.419 (Accepted)
950 -- 0.382 -> 0.361 (Accepted)
960 -- 0.424 -> 0.429 (Accepted)
970 -- 0.460 -> 0.493 (Rejected)
980 -- 0.477 -> 0.494 (Accepted)
990 -- 0.427 -> 0.460 (Accepted)
1000 -- 0.409 -> 0.450 (Accepted)

```

Now that you have a functioning MCMC program, play with it. Change the `printFrequency` to one so you can see every MCMC cycle. Examine the early output, before the chain has reached stationarity. Change the `numTosses` and `numHeads` values, making certain to keep `numTosses` \geq `numHeads`. Does doing this change the values visited by the chain?

7.4 Summarizing MCMC output

Programs that implement MCMC typically output the current state of the chain to a file. Often, they ‘thin’ the chain by only printing the chain’s state every so often. The problem with our program is that it does not remember where the chain has been. There is no (easy) way to make inferences about the posterior probability distribution of θ because we do not have the MCMC samples.

We will modify the program to do this. First, add the following snippet of code just before the `for` loop:

Example

```

// run the Markov chain
int bins[100];
for (int i=0; i<100; i++)
    bins[i] = 0;
int numSamples = 0;
for (int n=1; n<=chainLength; n++)
{
    // propose a new value for theta

```

Note that I also show some of the code that should already be in your `main` function as a reference point, so you know where to insert the new code. All we do is declare a vector of 100 `ints`, called `bins` and another `int` variable called `numSamples`. The idea is that the `bins` variable will count how often the chain visits states between $0.0 - 0.01$ (for `bins[0]`), $0.01 - 0.02$ (for `bins[1]`), $0.02 - 0.03$ (for `bins[2]`), and so forth. The other variable, `numSamples` will count how many samples were taken.

Next, add the following code after the second print statement, but before the main MCMC loop ends:

Example

```
// sample the chain
if (n % sampleFrequency == 0)
{
    numSamples++;
    bins[(int)(theta*100.0)]++;
}
```

After we finish the `for` loop, the `bins` vector will contain `numSamples` samples from the chain, sampled ever `sampleFrequency` MCMC cycles.

It would be nice to see the state of the `bins` variable after the chain has completed. In the following, I will provide code that constructs a nicely-formatted table. After the `for` loop, add the following code:

Example

```
// summarize the results
double cumulativeProbability = 0.0;
for (int i=0; i<100; i++)
{
    double intervalProbability = (double)bins[i] / numSamples;
    cumulativeProbability += intervalProbability;
    std::cout << std::fixed << std::setprecision(2) << i * 0.01 << " - ";
    std::cout << std::fixed << std::setprecision(2) << (i+1) * 0.01 << " -- ";
    std::cout << std::setw(5) << bins[i] << " ";
    std::cout << std::fixed << std::setprecision(3) << intervalProbability << " ";
    std::cout << std::fixed << std::setprecision(3) << cumulativeProbability << " ";
    std::cout << std::endl;
}

return 0;
```

I included the `return 0` statement as a reference to help you position this new code.

I changed the `chainLength` variable to be equal to 1000000 and then ran the program. A portion of the table summarizing the results looked like:

0.20 - 0.21 --	2	0.000	0.000
0.21 - 0.22 --	2	0.000	0.000
0.22 - 0.23 --	4	0.000	0.000
0.23 - 0.24 --	4	0.000	0.000
0.24 - 0.25 --	33	0.000	0.000
0.25 - 0.26 --	57	0.000	0.000
0.26 - 0.27 --	157	0.000	0.000
0.27 - 0.28 --	380	0.000	0.001
0.28 - 0.29 --	746	0.001	0.001
0.29 - 0.30 --	1384	0.001	0.003
0.30 - 0.31 --	2690	0.003	0.005
0.31 - 0.32 --	4464	0.004	0.010
0.32 - 0.33 --	7360	0.007	0.017
0.33 - 0.34 --	11589	0.012	0.029
0.34 - 0.35 --	17353	0.017	0.046
0.35 - 0.36 --	24714	0.025	0.071
0.36 - 0.37 --	33304	0.033	0.104
0.37 - 0.38 --	43109	0.043	0.147
0.38 - 0.39 --	53009	0.053	0.200
0.39 - 0.40 --	63120	0.063	0.263
0.40 - 0.41 --	71476	0.071	0.335
0.41 - 0.42 --	77570	0.078	0.413
0.42 - 0.43 --	80798	0.081	0.493
0.43 - 0.44 --	80446	0.080	0.574
0.44 - 0.45 --	77102	0.077	0.651
0.45 - 0.46 --	72045	0.072	0.723
0.46 - 0.47 --	63031	0.063	0.786
0.47 - 0.48 --	54271	0.054	0.840
0.48 - 0.49 --	44085	0.044	0.884
0.49 - 0.50 --	34516	0.035	0.919
0.50 - 0.51 --	26195	0.026	0.945
0.51 - 0.52 --	19101	0.019	0.964
0.52 - 0.53 --	13272	0.013	0.977
0.53 - 0.54 --	8980	0.009	0.986
0.54 - 0.55 --	5670	0.006	0.992
0.55 - 0.56 --	3352	0.003	0.995
0.56 - 0.57 --	2164	0.002	0.998
0.57 - 0.58 --	1118	0.001	0.999
0.58 - 0.59 --	675	0.001	0.999
0.59 - 0.60 --	354	0.000	1.000
0.60 - 0.61 --	171	0.000	1.000
0.61 - 0.62 --	63	0.000	1.000
0.62 - 0.63 --	43	0.000	1.000
0.63 - 0.64 --	12	0.000	1.000
0.64 - 0.65 --	6	0.000	1.000

(I didn't show the part of the table before 0.20 or after 0.65.) The first two values identify the interval; the next column is the count of the number of times the Markov chain visited the interval; the third column is the posterior probability of the interval, which is simply calculated as the fraction of the time the chain was in the interval; the last column is the cumulative probability up to the right edge of the interval.

Bayesians are religious about the idea that inferences should be based on the posterior probability distribution of the parameter. That said, they are fairly agnostic about how to summarize the posterior distribution once it's in hand. For example, one could form a point estimate by taking the mean or mode of the posterior probability distribution. A credible interval can be formed by examining the cumulative probability. Note that the 0.025 quantile is somewhere between 0.33 and 0.34. Similarly, the 0.975 quantile occurs between 0.52 and 0.53. Therefore, a rough 95% credible interval for the parameter θ is (0.33, 0.53). In words, there's a 95% probability that the true value lies in that interval. Interestingly, if you ask most scientists (excluding statisticians) to describe a confidence interval, they instead describe a credible interval!

Chapter 8

Representing Trees in Computer Memory

8.1 The basic idea

Figure 8.1a shows an example of a phylogenetic tree of three species. The species are named *Sp1*, *Sp2*, and *Sp3*. The branches of the tree have lengths, which are typically in terms of expected number of substitutions per site, but could also be in some other unit, such as time. The branch lengths are also indicated on the tree. A tree is a complex thing consisting of nodes and branches. The nodes are represented by the circles on the tree of Figure 8.1a whereas the branches are the

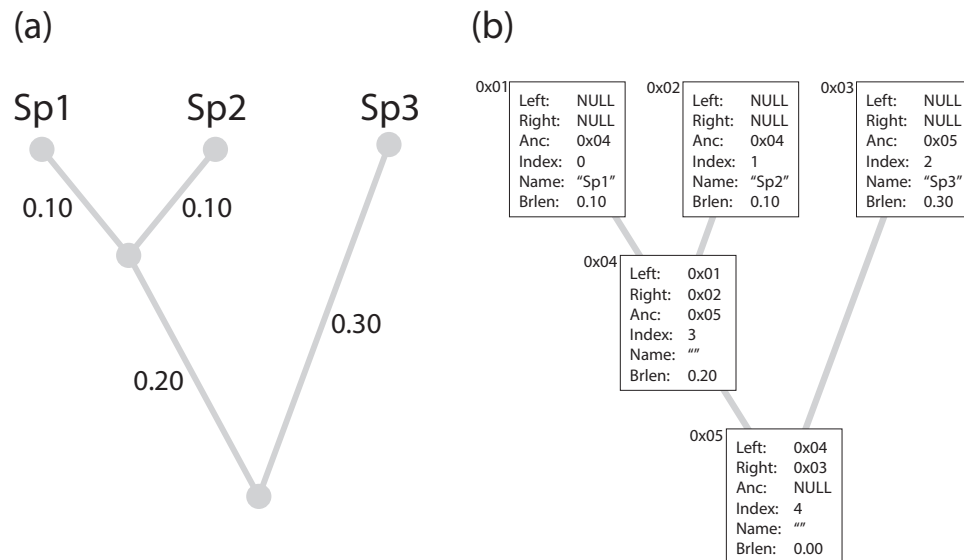


Figure 8.1: The nodes of a tree are objects, each of which contains information on its neighbors. The boxes represent the node object and the numbers next to each box, such as 0x01 represent the memory address of each object.

lines between the nodes. The tree of Figure 8.1a is rooted; there is a direction to tree with some nodes occurring before others¹. One of the nodes, the one furthest from the tips of the tree, is denoted the ‘root’ node.

How can we represent such a complicated object in computer memory? The idea is to represent a tree as a series of nodes, each of which has information on its neighbors. Figure 8.1b illustrates the basic idea. Each node on the tree is represented as a box in Figure 8.1b, but would be an object in C++. The address of each node is indicated next to each node in Figure 8.1b. You can see that the ancestor of the node with address 0x01, which represents Sp1, is the node with address 0x04, which happens to be the ancestor of Sp1 and Sp2.

The general strategy we will take in this chapter is to build up the machinery we need to represent and manipulate trees in computer memory. We will start with the implementation of a `Node` class, which will represent the nodes of the tree (*i.e.*, the boxes of Figure 8.1b). We will also implement a `Tree` class that will manage the nodes and allow us to manipulate the tree.

8.2 The Node class

Create a new project from your IDE. Name the project something sensible, like `ProjectTree` or `MyFirstTree`. Your new project will only consist of a single file with the `main` function. Now we need to add the basic files for our tree. First, create two new files from your IDE for a new C++ class called `Node`. One of the files should be named `Node.hpp` and the other should be named `Node.cpp`. While you are at it, create two new files called `Tree.hpp` and `Tree.cpp`. Now we have the files necessary to represent our trees. Let’s start coding!

Add the following code to the `Node.hpp` file:

Example

```
#ifndef Node_hpp
#define Node_hpp

#include <string>

class Node {

public:

    Node(void);
    Node*   getLft(void) { return left; }
    Node*   getRht(void) { return right; }
    Node*   getAnc(void) { return ancestor; }
```

¹The terminology I use to describe trees as consisting of ‘nodes’ and ‘branches’ is one that comes from the field of evolutionary biology. Mathematicians and computer scientists, however, see a tree as a type of graph, which has vertices and edges, which are equivalent to nodes and branches, respectively.

```

        int            getIndex(void) { return index; }
        std::string    getName(void) { return name; }
        double         getBranchLength(void) { return branchLength; }
        void           setLft(Node* p) { left = p; }
        void           setRht(Node* p) { right = p; }
        void           setAnc(Node* p) { ancestor = p; }
        void           setIndex(int x) { index = x; }
        void           setName(std::string s) { name = s; }
        void           setBranchLength(double x) { branchLength = x; }
        void           print(void);

protected:
    Node*              left;
    Node*              right;
    Node*              ancestor;
    int                index;
    std::string         name;
    double              branchLength;
};

#endif

```

Let's deconstruct this class definition. First, this class contains not one instance variable, as was the case for our `RandomVariable` class that only had a `int` `seed`, but six! We have three pointers of type `Node*`, an `int`, a `double`, and a `std::string`. The idea is that the `Node` pointers will hold the memory address of the neighbors to the node. The `index` variable will simply be a number that will help us distinguish nodes while debugging. The `branchLength` variable will hold the length of the branch for the node. We will follow the convention that a `Node`'s branch is below the `Node`. Finally, the `string` variable will hold the name of the node, if it happens to be a tip node that represents a living species. Because we are using the `std::string` type in this class, we have to include the `string` header file. You can see we do this before the class definition:

```
#include <string>
```

We made all of the instance variables for this class `protected`; other classes cannot directly access these variables, even if they have a reference to the node. We allow other parts of the code to change the instance variables with `public` functions. Each of these functions starts with `set` (e.g., `setAnc`, which sets the `ancestor` variable). Similarly, we provide functions that start with `get` that will return the current value of the instance variables. The `set` and `get` functions are called 'setters' and 'getters' among computer scientists.

You will note that not only did we declare the functions in this class, but we also implemented them at the same time. Consider the following line from our `Node` class definition:


```
void    setBranchLength(double x) { branchLength = x; }
```

We declare the function with `void setBranchLength(double x)`. However, we then go ahead and implement the function by continuing with the following snippet of code `{ branchLength = x; }`. This is counter to what I said earlier when we implemented the `RandomVariable` class. There, I said that you should implement the guts of the class in the implementation file (the file that ends `.cpp`). Often, programmers will do what we did here and implement the function in the header file. They typically do this if the function is a small one. Here, every one of the getters and setters has one line of code. I went ahead and implemented the getters and setters in the header file to save the time of doing so in the implementation file. You should try to implement one of the getters or setters in the implementation file to convince yourself that (1) you can do so and (2) that the program runs just fine if you do that.

We also declare a default constructor in the class definition, `Node(void)`, and a function named `print(void)`. Both of these functions will consist of more than a few lines of code, so we *will* implement them in the `Node.cpp` file. Add to the `Node.cpp` file the following code:

Example

```
#include "Node.hpp"

Node::Node(void) {

    left      = NULL;
    right     = NULL;
    ancestor  = NULL;
    index     = 0;
    name      = "";
    branchLength = 0.0;
}

void Node::print(void) {

    std::cout << "Node " << index << " (" << this << ")" << std::endl;
    std::cout << "    Lft:  " << left << std::endl;
    std::cout << "    Rht:  " << right << std::endl;
    std::cout << "    Anc:  " << ancestor << std::endl;
    std::cout << "    Name: \"\" << name << \"\" << std::endl;
    std::cout << "    Brlen: \"\" << branchLength << std::endl;
}
```

Our default constructor simply initializes all of the instance variables. Remember, the default constructor is called when we instantiate a `Node`. This means that when we first use a `Node` in the code that we can count on all of the variables starting with the values we initialize in the default constructor.

The `print` function simply prints to the console the state of the node. Two aspects of this function should stand out. First, we use the `this` keyword. Every object in C++ knows its own address, which can be accessed using `this`. A mildly interesting aspect of this function is that we print quotation marks from within a string. We do this using `\`.

8.3 The Tree class

The `Tree` class will manage the nodes. Because we do not know how big our tree will be, the `Tree` class will be responsible for allocating and deleting the `Node` objects that actually represent the tree structure and information.

In the `Tree.hpp` file, add the following code:

Example

```
#ifndef Tree_hpp
#define Tree_hpp

#include <vector>
class Node;

class Tree {

public:
    Tree(void);
    ~Tree(void);
    void listNodes(void);

protected:
    Node* root;
    std::vector<Node*> nodes;
};

#endif
```

In many respects, this class should look familiar to you. As you scan through the class definition, you should be comfortable with the idea that the class is defined as:

```
class Tree {

};
```

Seeing variables that are `public` and `protected` shouldn't freak you out. The idea of a instance variable that is a pointer, such as `Node* root`, should make you feel warm and fuzzy. That said, there are a few aspects of this class definition that are new. We will go through those one at a time.

One of the unfamiliar instance variables is:

```
std::vector<Node*> nodes;
```

This declares a vector of `Node` pointers called `nodes`. This is the first time you have brushed shoulders with the Standard Template Library (STL) in C++. The STL is a collection of tools available to the C++ programmer that include containers, algorithms, and iterators. The `vector` we use in the `Tree` class is an example of a container. As objects, STL containers have functions associated with them. For example, the `vector` container has two functions that we will take advantage of: `push_back` and `size`. The `push_back` function adds a new thing-to-be-contained to the end of the vector object. The `size` function returns the number of 'things' in the vector. In the `Tree` class, the 'things' in the vector are `Node` pointers. When we declare the `vector` object, we also indicate what type of thing will be contained in it. We do this with the `<` and `>` signs: `vector<Node*>`. Note that we need to include the file `vector` if we are to use a `vector` in our code. This explains the compiler directive, `#include <vector>`.

You will also note that before the class definition, we added the line:

```
class Node;
```

This is called 'forward declaration.' We forward declare `Node` because we make reference to a `Node` pointer in the class definition. (In fact, we do this twice, once when we declare the `root` variable and again when we declare the `nodes` variable.) We could also have simply included the `Node.hpp` file, right before or after we include the `vector` header,

```
#include <vector>
#include "Node.hpp"
```

in which case we could forgo the statement `class Node`. Both are equivalent in as much as the code will compile and run. However, in keeping with the principle of including the least amount of information necessary, the method employed here is preferable. In the class definition, we only declare `Node` pointers (`Node*`). All pointers take up the same amount of memory. Essentially, all the compiler has to do when examining this file is trust that we have implemented the `Node` class elsewhere.

The last unfamiliar element of the `Tree` class is `~Tree(void)`. This looks a lot like our default constructor, except for the addition of the tilde symbol. The addition of the tilde indicates that this is a destructor function. As mentioned earlier, constructor functions are called when an object is created. The destructor function, as the name implies, is called just before the object is destroyed. Constructors are useful places to carry out initialization. Destructors, by contrast, are an excellent place to carry out clean-up activities just before an object is destroyed.

Let's implement the `Tree` class. Add code to the `Tree.cpp` file so it reads:

Example

```
#include <iostream>
#include "Node.hpp"
#include "Tree.hpp"

Tree::Tree(void) {

    // make the three-species tree of Figure 8.1

    // allocate the five nodes for the three-species tree
    for (int i=0; i<5; i++)
    {
        Node* newNode = new Node;
        nodes.push_back( newNode );
    }
    for (int i=0; i<nodes.size(); i++)
        std::cout << "nodes[" << i << "] = " << nodes[i] << std::endl;

    // set the member pointers
    Node* p = nodes[0];
    p->setAnc(nodes[3]);
    p->setLft(NULL);
    p->setRht(NULL);
    p->setIndex(0);
    p->setBranchLength(0.10);
    p->setName("Sp1");

    p = nodes[1];
    p->setAnc(nodes[3]);
    p->setLft(NULL);
    p->setRht(NULL);
    p->setIndex(1);
    p->setBranchLength(0.10);
    p->setName("Sp2");

    p = nodes[2];
    p->setAnc(nodes[4]);
    p->setLft(NULL);
    p->setRht(NULL);
    p->setIndex(2);
```

```

    p->setBranchLength(0.30);
    p->setName("Sp3");

    p = nodes[3];
    p->setAnc(nodes[4]);
    p->setLft(nodes[0]);
    p->setRht(nodes[1]);
    p->setIndex(3);
    p->setBranchLength(0.20);
    p->setName("");

    p = nodes[4];
    p->setAnc(NULL);
    p->setLft(nodes[3]);
    p->setRht(nodes[2]);
    p->setIndex(4);
    p->setBranchLength(0.0);
    p->setName("");

    root = nodes[4];
}

Tree::~~Tree(void) {

    for (int i=0; i<nodes.size(); i++)
        delete nodes[i];
}

void Tree::listNodes(void) {

    for (int i=0; i<nodes.size(); i++)
    {
        nodes[i]->print();
    }
}

```

This code sacrifices brevity in order to be more transparent. We will walk through this code function-by-function.

Tree(void) — The default constructor is called when we instantiate a **Tree**. (A tree would be instantiated with the declaration, **Tree myTree;**.) The default constructor makes the tree shown

in Figure 8.1. This tree has three tips and five nodes, so the first thing we do in the default constructor is allocate five Nodes:

```
// allocate the five nodes for the three-species tree
for (int i=0; i<5; i++)
{
    Node* newNode = new Node;
    nodes.push_back( newNode );
}
for (int i=0; i<nodes.size(); i++)
    std::cout << "nodes[" << i << "] = " << nodes[i] << std::endl;
```

Within the loop where we allocate the nodes we do two things. First, we declare a **Node** pointer variable called **newNode** and set it equal to the memory address of the new node that was allocated using the **new** function. The **new** function allocates the **Node** object and then returns the memory address of the just-allocated object. We need to capture that address, and do so with the **newNode** variable. We then add the memory address of the new **Node** to the **nodes** vector using the **push_back** function of the **vector** object. After the nodes are allocated, their memory addresses are printed to the console.

The following lines of code are more interesting because you can see the real action for making the tree. We set the information for each node that was allocated, one at a time. For example, the first block of seven lines of code,

```
Node* p = nodes[0];
p->setAnc(nodes[3]);
p->setLft(NULL);
p->setRht(NULL);
p->setIndex(0);
p->setBranchLength(0.10);
p->setName("Sp1");
```

sets the information for the first node. For convenience, we declare a **Node** pointer variable called **p** and set it equal to the memory address of the first node we allocated, **Node* p = nodes[0]**. In the next line of code, we set the **ancestor** instance variable of **nodes[0]** to be equal to **nodes[3]**. Because the node pointed to by **p** is a tip, we set its **left** and **right** instance variables to be **NULL**. Next, we set the **index**, **branchLength**, and **name** instance variables of **p** by calling the appropriate setters.

In the earlier examples, we used the dot (‘.’) notation to call an object’s function. Here, we are using the arrow (‘->’) operator to call an object’s function. How do you know when to use the dot operator versus the arrow operator? The rule: use the dot operator when you have the object and the arrow operator when you have a pointer to the object. In this code, we have pointers (memory addresses) to **Nodes**, so we use the arrow notation.

The other blocks of code set up the information for the remaining nodes of the tree of Figure 8.1. The very last statement in the default constructor sets the **root** instance variable of the **Tree** class to be equal to the memory address contained at **nodes[4]**.

~Tree(void) — In the constructor, we allocated **Nodes** that constitute the tree. In the destructor, we free that memory using the **delete** function. If we were to neglect doing this, we would

have introduced a memory leak bug to the program. Whenever you allocate memory with the `new` function, you should remember to free that memory with the `delete` function. We dynamically allocated memory once before, where we allocated an array of `int` variables:

```
int numToAllocate = 1000;
int* x = new int[numToAllocate];
```

which should be freed later in the code:

```
delete [] x;
```

Note that the `delete` function was followed by open and closed square brackets (`[]`). In the `Tree` class destructor, however, we don't follow the `delete` function with the square brackets. Why? You should include the square brackets after the `delete` function when you allocated an array of objects. If you only allocate a single object or variable, then you do not use the square brackets. The following code snippet demonstrates the idea:

```
int* x = new int[100]; // allocate an array
int* y = new int;      // allocate a single int variable

// do some stuff to x and y

delete [] x;           // use the brackets, [], because x is an array
delete y;              // ixnay on the racketbray because y is a single element
```

`listNodes(void)` — This function simply loops over the vector, `nodes`, and calls the `print` function on each. This allows us to see the information on the tree.

Before we can check the correctness of our `Tree` implementation, we need to make an instance of a tree. Change the `main` function of the program to actually instantiate a tree:

Example

```
#include "Tree.hpp"

int main(int argc, char* argv[]) {

    Tree t;
    t.listNodes();

    return 0;
}
```

The `main` function declares a `Tree` variable called `t` (*i.e.*, we instantiate one instance of the `Tree` class). We then call the `listNodes` function. When I run the program, the output looked like:

```

nodes[0] = 0x10040d030
nodes[1] = 0x10040d070
nodes[2] = 0x10040c480
nodes[3] = 0x10040c4c0
nodes[4] = 0x10040c500
Node 0 (0x10040d030)
  Lft: 0x0
  Rht: 0x0
  Anc: 0x10040c4c0
  Name: "Sp1"
  Brlen: 0.1
Node 1 (0x10040d070)
  Lft: 0x0
  Rht: 0x0
  Anc: 0x10040c4c0
  Name: "Sp2"
  Brlen: 0.1
Node 2 (0x10040c480)
  Lft: 0x0
  Rht: 0x0
  Anc: 0x10040c500
  Name: "Sp3"
  Brlen: 0.3
Node 3 (0x10040c4c0)
  Lft: 0x10040d030
  Rht: 0x10040d070
  Anc: 0x10040c500
  Name: ""
  Brlen: 0.2
Node 4 (0x10040c500)
  Lft: 0x10040c4c0
  Rht: 0x10040c480
  Anc: 0x0
  Name: ""
  Brlen: 0
Program ended with exit code: 0

```

Yay! We now have a three-species tree in computer memory. You should scrutinize the output from your implementation to convince yourself that the information printed for each node is, indeed, equivalent to the tree shown in Figure 8.1.

8.4 Traversing trees in pre- or postorder

As thrilling as the idea of having a tree in computer memory is (and it is pretty cool), so what? If you think about it, it's rather useless to have a tree in computer memory if you can't actually *do* anything with that representation.

Many of the algorithms we use in evolutionary biology require that the nodes of the tree are traversed in order, from the tips to the root or from the root to the tips (*e.g.*, we traverse the tree from the tips to the root when using the Felsenstein pruning algorithm; Felsenstein, 1981; Gallager, 1962, 1963). A traversal from the tips to the root is called a postorder traversal of the tree. The opposite, a traversal from the root to the tips, is called a preorder traversal.

One of the grooviest aspects of representing trees as a linked list of pointers is that you can apply recursive algorithms to get the pre- or postorder traversal sequence for the tree. How do these algorithms work?

Modify a portion of the `Tree.hpp` file so that it reads:

Example

```
#ifndef Tree_hpp
#define Tree_hpp

#include <vector>
class Node;

class Tree {

public:
    Tree(void);
    ~Tree(void);
    std::vector<Node*>& getTraversalOrder(void) { return postOrderSequence; }
    void listNodes(void);

protected:
    void initializeTraversalOrder(void);
    void passDown(Node* p);
    Node* root;
    std::vector<Node*> nodes;
    std::vector<Node*> postOrderSequence;
};

#endif
```

We added another STL vector called `postOrderSequence` which will eventually hold the memory addresses in the postorder traversal order. We also declared and implemented a public getter function that returns this sequence. Finally, we declare two new functions, `initializeTraversalOrder` and `passDown`, which will be implemented in the `Tree.cpp` file. Modify the implementation file, `Tree.cpp`, by adding two functions:

Example

```
void Tree::initializeTraversalOrder(void) {

    postOrderSequence.clear();
    passDown(root);
}

void Tree::passDown(Node* p) {

    if (p != NULL)
    {
        passDown(p->getLft());
        passDown(p->getRht());
        postOrderSequence.push_back(p);
    }
}
```

Finally, call the `initializeTraversalOrder` function on the last line of the default constructor:

Example

```
    root = nodes[4];
    initializeTraversalOrder();
}
```

The `initializeTraversalOrder` function does two things. First, it makes certain that the vector `postOrderSequence` is empty by calling the `clear` function. The `clear` function is another function that is part of the vector implementation that empties out a vector. Once we are certain that the `postOrderSequence` vector is empty, we call another function, `passDown`. We pass the `root` variable (the memory address of the root) to the `passDown` function. This all seems quite straight forward.

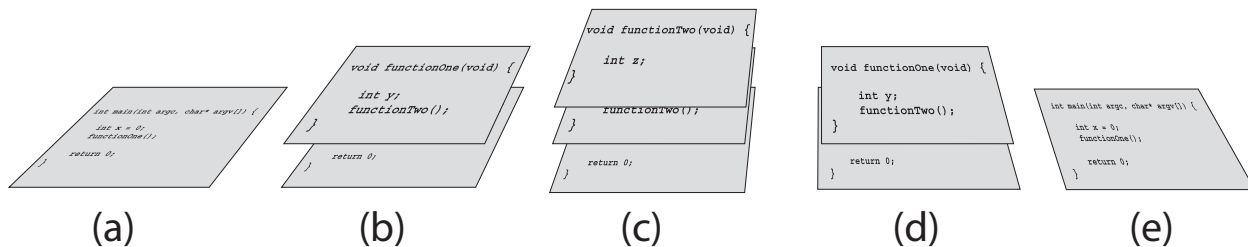


Figure 8.2: The call stack for the simple program with three functions. (a) `main` is the first function laid on top of the stack. (b) `main` calls `functionOne`, which is then put on top of the call stack. (c) `functionOne` calls `functionTwo`, which is put on the call stack, but (d) removed once it finishes execution. `functionOne` finishes execution and is removed from the call stack, leaving (e) the `main` function to resume, and complete, execution.

It's in the `passDown` function that things get interesting. First, we check that the `Node` address that is passed into the function through the variable `p` is not `NULL`. If it is not `NULL`, then we call `passDown` again, this time with the memory address of the node to the left of `p`. A function calling itself? Mind blown! As if to compound the craziness, we call `passDown` again, this time with the memory address of the node to the right of `p`. It's only after these two recursive function calls that we add `p` to the vector holding the postorder traversal sequence.

The first time I saw the `passDown` function, I (almost) believed that magic was involved. Of course, I regained my composure: "This is a computer," I told myself, "A totally deterministic machine. Something sensible must be occurring in memory." I was right². Something sensible is occurring.

When your program is run, the current location of the execution of the code is maintained by something called the 'call stack.' Consider the following simple program:

```
void functionOne(void);
void functionTwo(void);

int main(int argc, char* argv[]) {

    int x = 0;
    functionOne();
    return 0;
}

void functionOne(void) {

    int y;
    functionTwo();
}
```

²These rational thoughts probably explain why I've had a career in science and not in palmistry.

```
void functionTwo(void) {

    int z;
}
```

When the program executed, it starts at `main` and executes the instruction, to set aside memory for a variable called `x`, first. It then calls `functionOne`. This function is then put on the top of the call stack and executed one line at a time. It first sets aside memory for an `int` variable called `y` (within the scope of that function), then calls `functionTwo`, which is then put on top of the call stack. Execution of `functionTwo` then begins. A variable is declared, but then destroyed when we reach the end of `functionTwo`. The key point: When execution of `functionTwo` completes, it is removed from the top of the call stack. We return to the point of `functionOne` just after `functionTwo` was called. `functionOne` then completes execution, the variable `y` is destroyed, and it is removed from the top of the call stack. We return to the `main` function, and exit `main`, returning the number 0 as we do so. Figure 8.2 illustrates the idea.

To fully understand the `passDown` function, I encourage you to try the following fun exercise. Make a document that contains only the `passDown` function. Make certain that the font size is large, taking up as much of the page as possible, without causing the individual lines to wrap around. Center the function on the page and print about 10 copies of the page. Look at Figure 8.1b and imagine that you are calling `passDown` from the `initializeTraversalOrder` function, which is called with the memory address of the `root` node of the tree. Take a clean sheet of paper with the printed function on it and with a pencil, note the memory address that is passed into `passDown` the first time. Begin tracing the execution of that function, line-by-line. Every time you recursively call `passDown`, add a fresh sheet of paper on top of the first, noting again with a pencil the memory address that is passed into the `passDown` function. As you traverse the tree, your stack of papers will grow and shrink. When you reach a tip node, in which both the `left` and `right` memory address are `NULL`, you will add another sheet of paper, but this time you will pass `NULL` into the function. Note that the recursive function calls will be completely bypassed. Once you reach the end of execution for a `passDown` function call, remove the paper from the stack. You should end up with a clean desk.

Does the routine work? Let's find out. Modify your `main` function to read:

Example

```
#include <iostream>
#include "Node.hpp"
#include "RandomVariable.hpp"
#include "Tree.hpp"

int main(int argc, char* argv[]) {
```

```
Tree t;

std::vector<Node*> postOrd = t.getTraversalOrder();

std::cout << "Postorder: ";
for (int i=0; i<postOrd.size(); i++)
    std::cout << postOrd[i]->getIndex() << " ";
std::cout << std::endl;

std::cout << "Preorder: ";
for (int i=postOrd.size()-1; i>=0; i--)
    std::cout << postOrd[i]->getIndex() << " ";
std::cout << std::endl;

return 0;
}
```

We get a reference to the postorder traversal sequence from the tree by calling the `t.getTraversalOrder` function. We then print the `vector` that is returned twice; once from the beginning to the end (*i.e.*, in postorder) and then from the end to the beginning (*i.e.*, in preorder). When I run the program, I get the following output:

```
Postorder: 0 1 3 2 4
Preorder: 4 2 3 1 0
Program ended with exit code: 0
```

The nodes are being visited in the correct order!

Chapter 9

Simulating the Birth-Death Process of Cladogenesis

9.1 A stochastic model of cladogenesis

The birth-death process of cladogenesis is widely-used in evolutionary biology to model the Tree of Life. The process models the splitting (speciation) and termination (extinction) of lineages through time. The probability of a speciation event occurring in an infinitesimal interval of time, Δt , is $\lambda\Delta t$ whereas the probability of an extinction event occurring in the same infinitesimal interval of time is $\mu\Delta t$. This implies that the time between speciation *or* extinction events is exponentially distributed with parameter $\lambda + \mu$. When an event occurs, it is a speciation with probability $\frac{\lambda}{\lambda+\mu}$ or an extinction with probability $\frac{\mu}{\lambda+\mu}$.

Kendall (1948) derived the probability that a single lineage, extant at time t in the past, gives rise to N lineages in the present:

$$P_N(t) = \left(\frac{\lambda}{\mu}\right)^{N-1} P_1(t) [P_0(t)]^{N-1}$$

for $N \geq 2$, where $P_0(t)$ is the probability that the process dies out, leaving no species alive at the end of the process, and is equal to

$$P_0(t) = \frac{\mu - \mu e^{-(\lambda-\mu)t}}{\lambda - \mu e^{-(\lambda-\mu)t}}$$

and $P_1(t)$ is the probability that exactly one lineage survives

$$P_1(t) = \frac{(\lambda - \mu)^2 e^{-(\lambda-\mu)t}}{(\lambda - \mu e^{-(\lambda-\mu)t})^2}.$$

The quantities $P_0(t)$ and $P_1(t)$ are particularly important because they are used as the building blocks to compute the probability of a time-calibrated phylogenetic tree (Thompson, 1975). Kendall (1948) also derived the expected number of lineages alive after a duration of t : $E(N) = N_0 e^{(\lambda-\mu)t}$, where N_0 is the number of lineages at time $t = 0$.

Table 9.1 gives the probability of N species surviving to the present when $\lambda = 3$, $\mu = 1$, and $t = 1$. Note that for this choice of parameter values, the process is expected to die out, leaving no

Table 9.1: Probabilities of N species surviving to the present when $\lambda = 3$, $\mu = 1$, and the simulations are run over a duration $t = 1$.

N	$P_N(t)$	N	$P_N(t)$	N	$P_N(t)$	N	$P_N(t)$	N	$P_N(t)$
0	0.301838	5	0.044351	10	0.027001	15	0.016438	20	0.010008
1	0.065967	6	0.040161	11	0.024450	16	0.014885	> 20	0.095909
2	0.059734	7	0.036366	12	0.022140	17	0.013479		
3	0.054090	8	0.032930	13	0.020049	18	0.012205		
4	0.048979	9	0.029818	14	0.018154	19	0.011052		

living descendants, about 30% of the time. The growing evolutionary tree is fragile when there are only a few lineages. A string of bad luck can wipe out all of the lineages, terminating the process.

Figure 9.1 shows 64 realizations of the birth-death process when $\lambda = 3$, $\mu = 1$, and $t = 1$. Note the variability in the realized outcomes of the process. Do the simulated results look like genuine evolutionary trees to you? What simplifying assumptions are being made that make you uncomfortable?

9.2 Coding the birth-death process

We will modify the project from the previous chapter, in which you implemented `Tree` and `Node` classes. We know that we will need to generate random numbers to generate a birth-death tree. Therefore, add the `RandomVariable` class you made earlier to this project. (In XCode, go to the `File` menu and select `File > Add Files to "<Project Name>"...` Select the files from your earlier project. You should select the option to copy the files into your new project directory.)

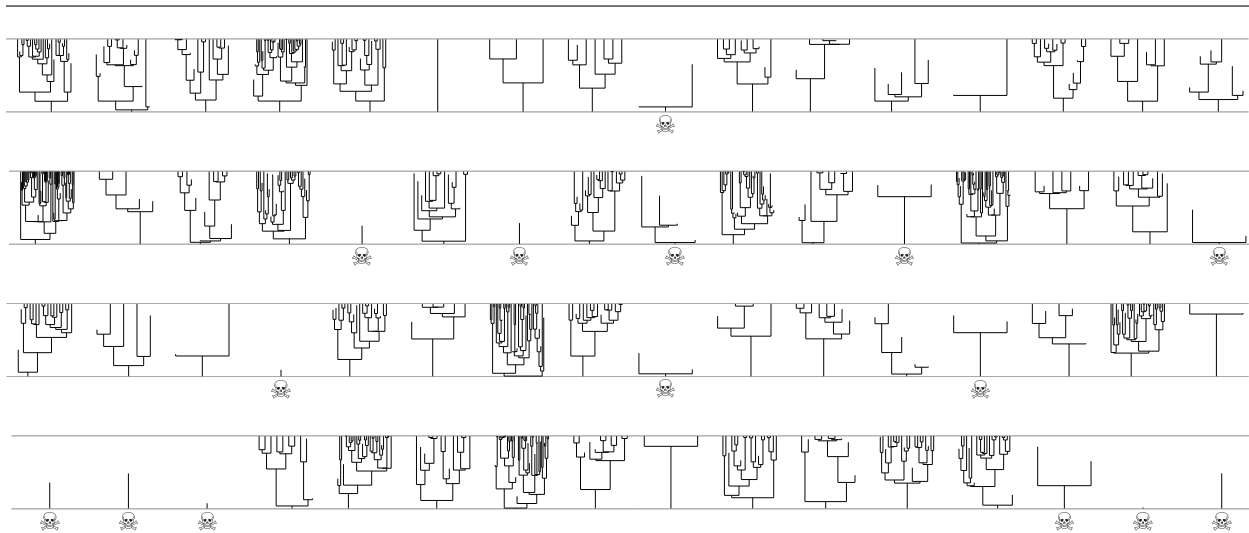


Figure 9.1: Realizations of the birth-death process of cladogenesis.

There are multiple places we could implement the algorithm for building the birth-death process tree. We will build the tree in the `Tree` constructor. First, delete the default constructor from the implementation file of the original project. Also, change the `Tree.hpp` file such that it reads:

Example

```
#ifndef Tree_hpp
#define Tree_hpp

#include <vector>
class Node;
class RandomVariable;

class Tree {

public:
    Tree(double lambda, double mu, double duration,
          RandomVariable* rv);
    ~Tree(void);
    std::vector<Node*>& getTraversalOrder(void) { return postOrderSequence; }
    void listNodes(void);

protected:
    Tree(void) { }
    void initializeTraversalOrder(void);
    void passDown(Node* p);
    Node* root;
    std::vector<Node*> nodes;
    std::vector<Node*> postOrderSequence;
};

#endif
```

We removed the default constructor from the `Tree.cpp` file, but reimplemented it in the header file. Where did we do this? Look at the first line under the key word `protected`. Not only do we declare the default constructor there, but we implement it between the curly brackets: `Tree(void) { }`. This means that we will no longer be able to instantiate a `Tree` object using the default constructor. Doing so causes an error, preventing the program from being compiled. (In XCode, the error reads, `Calling a protected constructor of class 'Tree'.`) We are forcing ourselves, and others, to use our new constructor to instantiate a tree. This is a clever use of the `protected` (or `private`) key words in C++.

Now, modify your `Tree.cpp` file so that it looks like:

Example

```

#include <iostream>
#include "Node.hpp"
#include "RandomVariable.hpp"
#include "Tree.hpp"

Tree::Tree(double lambda, double mu, double duration, RandomVariable* rv) {

    // generate a birth-death with parameter lambda, mu, and duration
    std::cout << "Calling the birth-death tree constructor" << std::endl;

}

Tree::~Tree(void) {

    std::cout << "Calling the Tree destructor with " << nodes.size() << " nodes" << std::endl;
    for (int i=0; i<nodes.size(); i++)
        delete nodes[i];
}

void Tree::initializeTraversalOrder(void) {

    postOrderSequence.clear();
    passDown(root);
}

void Tree::listNodes(void) {

    for (int i=0; i<nodes.size(); i++)
    {
        nodes[i]->print();
    }
}

void Tree::passDown(Node* p) {

    if (p != NULL)
    {
        passDown(p->getLft());
    }
}

```

```
        passDown(p->getRht());
        postOrderSequence.push_back(p);
    }
}
```

Finally, modify your `main` function:

Example

```
#include "RandomVariable.hpp"
#include "Tree.hpp"

int main(int argc, char* argv[]) {

    RandomVariable rv;

    Tree t(3.0, 1.0, 1.0, &rv);
    t.listNodes();

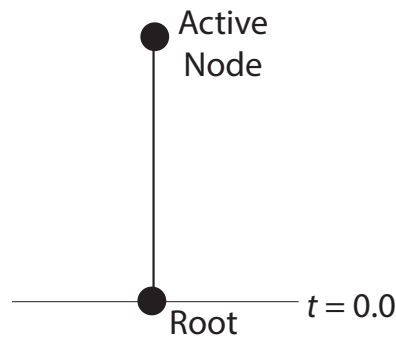
    return 0;
}
```

This program should compile and run. Try to do so. When I run the program, the output is:

```
Calling the birth-death tree constructor
Calling the Tree destructor with 0 nodes
Program ended with exit code: 0
```

This output confirms that the new `Tree` constructor is called and that when the tree variable, `t`, goes out of scope in `main` that the destructor is called. Now that you have confirmed that everything is working as we expect, remove the print statements in the new constructor and the destructor.

We will start our tree with two `Nodes`. One will be the root and the other node its descendant. Because the root node has a descendant, it cannot itself speciate or go extinct. Its fate has been determined. The other node, however, we might designate the ‘active node’ because it can speciate or go extinct; its fate has not yet been determined. A simple method for generating a birth-death tree is to keep a list of active nodes. We start the process with a single active node. You can visualize the starting condition for the tree:



To implement this beginning state, enter the following code in the new `Tree` constructor:

Example

```
Tree::Tree(double lambda, double mu, double duration, RandomVariable* rv) {

    // generate a birth-death with parameter lambda, mu, and duration

    // initialize the single lineage, adding
    // the descendant to a list of active nodes
    nodes.push_back( new Node );
    nodes.push_back( new Node );
    nodes[0]->setLft(nodes[1]);
    nodes[1]->setAnc(nodes[0]);
    std::set<Node*> activeNodes;
    activeNodes.insert( nodes[1] );
}
```

We make use of another type of STL container in this snippet of code. A `set` in C++ acts just as a set does in mathematics. We use a set because it's easy to insert and delete elements from the set.

Now for the fun part. We will add code that represents the skeleton for generating a birth-death process tree:

Example

```
Tree::Tree(double lambda, double mu, double duration, RandomVariable* rv) {
```


I'm an infinite loop. Wheeeeeeeeeee!
 Program ended with exit code: 9

Modify the while loop to fix the problem and add some code for generating our tree:

Example

```
// generate the full tree
double t = 0.0;
while (t < duration)
{
    // increment t using the exponential distribution
    double rate = activeNodes.size() * (lambda + mu);
    t += rv->exponentialRv(rate);

    // if t is still less than duration, go ahead do the speciation
    // or extinction thing on a randomly selected active lineage
    if (t < duration)
    {
        // choose a node
        Node* p = NULL;

        // choose a type of event
        double u = rv->uniformRv();
        if ( u < lambda / (lambda+mu) )
        {
            // speciation event

        }
        else
        {
            // extinction event

        }
    }
}
```

Now, the logic of the process is fully exposed, even if we haven't completed the implementation. Each time we go through the loop, we increment t by adding to it an exponentially-distributed random variable. The parameter of the exponential distribution is the overall rate of change. For a single lineage, the overall rate at which a speciation or extinction event occurs is $\lambda + \mu$. The overall rate, then, is n times greater if there are n lineages. After all, each lineage has a rate

of speciation and extinction. The set of `activeNodes` contains all those nodes that can speciate or go extinct. The overall rate, then, is simply `activeNodes.size() * (lambda + mu)`. The next line of code increments `t` by the exponential random variable. The operator `+=` adds to the original value. The two statements, `x = x + 3` and `x += 3` are equivalent in C++.

If after we increment, the variable `t` remains less than `duration`, we know that we will subject one of the nodes in the list `activeNodes` to a speciation or extinction event. We need to do two things. First, we need to randomly select one of the nodes in `activeNodes`. We also need to decide whether it is a speciation or extinction event. We have only partially implemented both.

First, add a new function to the `Tree` class that will choose a `Node*` at random from the `set` of `activeNodes`. The implementation of the function should be:

Example

```
Node* Tree::chooseNodeFromSet(std::set<Node*>& s, RandomVariable* rv) {

    int whichNode = (int)(s.size() * rv->uniformRv());
    int i = 0;
    for (Node* nde : s)
    {
        if (whichNode == i)
            return nde;
        i++;
    }
    return NULL;
}
```

The STL `set` has easy insertion and deletion, but because of the internal representation of the elements in a `set`, it is difficult to randomly pick any element from the set. Here, we choose a random integer between 0 and the number of elements in the set, minus one. We then loop over the elements in the set until we hit the element whose offset is the same as our random integer. There are better ways to do this, but none that I could think of that were as clear.

Modify the code in the loop as follows:

Example

```
if (t < duration)
{
    // choose a node
```

```

Node* p = chooseNodeFromSet(activeNodes, rv);

// choose a type of event
double u = rv->uniformRv();
if ( u < lambda / (lambda+mu) )
{
    // speciation event
    Node* newLft = new Node;    // 1. allocate new left and
    Node* newRht = new Node;    //    right nodes
    nodes.push_back(newLft);    // 2. add the new nodes to the tree
    nodes.push_back(newRht);
    newLft->setAnc(p);           // 3. set the ancestor of both new
    newRht->setAnc(p);           //    nodes to be p
    p->setLft(newLft);           // 4. set the left and right values of
    p->setRht(newRht);           //    p to be the new nodes
    activeNodes.erase(p);       // 5. modify the list of active nodes
    activeNodes.insert(newLft);
    activeNodes.insert(newRht);
}
else
{
    // extinction event
    activeNodes.erase(p);       // poor p ... he's dead
}
}

```

Your program now generates a tree under the birth-death process of cladogenesis. Unfortunately, the output is less than inspiring. For one, we do not track when the speciation or extinction events occur, so it's impossible to report the branch lengths of the tree. Let's spend some time cleaning up our implementation.

The first task is to add to the `Node` class a new variable that will keep track of the time for the node. I am going to leave it up to you to do this without holding your hand. Just make certain that the time variable is stored as a `double` and that the getter and setter are called `getTime` and `setTime`. In the constructor for the `Tree` class, add the following line:

Example


```
// choose a node
Node* p = chooseNodeFromSet(activeNodes, rv);
p->setTime(t);
```

The first two lines of code should already exist; they are there only to help you position the single line of new code: `p->setTime(t)`. When we choose `p`, we know its time will be fixed to the time of the event. Once we know `p`, we might as well set its time using the setter you implemented in the `Node` class.

Add a bit of code after the `while` loop:

Example

```
// clean up
numExtant = activeNodes.size();
for (Node* nde : activeNodes)
{
    nde->setTime(duration);
}
initializeTraversalOrder();

// set the index variable and assign branch lengths from the node times
int nodeIdX = 0;
for (int i=0; i<postOrderSequence.size(); i++)
{
    Node* p = postOrderTraversalOrder[i];
    if (p->getLft() == NULL && p->getRht() == NULL)
    {
        p->setIndex(nodeIdX++);
        p->setName( std::to_string(nodeIdX) );
    }
    if (p->getAnc() != NULL)
        p->setBranchLength( p->getTime() - p->getAnc()->getTime() );
}
for (int i=0; i<postOrderTraversalOrder.size(); i++)
{
    Node* p = postOrderTraversalOrder[i];
    if ( !(p->getLft() == NULL && p->getRht() == NULL) )
```

```

        p->setIndex(nodeIdx++);
    }

```

The code we just added does several things:

1. We assign the number of surviving nodes in `activeNodes` to the variable `numExtant`. If you try to compile this modified code, you'll get an error stating that there is no `numExtant` variable in the `Tree` class. Go ahead and add that instance variable to the tree. While you're at it, add a getter and setter for `numExtant`. What type of variable do you think this should be? A `double`? A `bool`? An `int`?
2. We set the times for any nodes that happen to have survived to the end by looping over the surviving nodes in `activeNodes` and assigning the time for each to be equal to `duration`.
3. We initialize the postorder traversal sequence for the tree.
4. We calculate the branch lengths for all of the nodes using the time of each node. Branch lengths on this tree will be in units of time.
5. While we are assigning the branch lengths, we also set the `index` instance variable for each node of the tree. Nodes are indexed in such a way that the tips of the tree are numbered $0, 1, \dots, N - 1$ and the interior nodes are indexed $N, N + 1, \dots$

Run the program. You should get a list of nodes for a tree that is a realization of the birth-death process. When I run the program, I obtained the following:

```

Node 7 (0x100658b60)
  Lft: 0x100603a20
  Rht: 0x0
  Anc: 0x0
  Name: ""
  Brlen: 0
  Time: 0
Node 6 (0x100603a20)
  Lft: 0x100603a70
  Rht: 0x1006594a0
  Anc: 0x100658b60
  Name: ""
  Brlen: 0.194768
  Time: 0.194768
Node 5 (0x100603a70)
  Lft: 0x100659520
  Rht: 0x100659570

```

```
Anc: 0x100603a20
Name: ""
Brlen: 0.311997
Time: 0.506765
Node 3 (0x1006594a0)
  Lft: 0x0
  Rht: 0x0
  Anc: 0x100603a20
  Name: "4"
  BrLen: 0.805232
  Time: 1
Node 0 (0x100659520)
  Lft: 0x0
  Rht: 0x0
  Anc: 0x100603a70
  Name: "1"
  BrLen: 0.493235
  Time: 1
Node 4 (0x100659570)
  Lft: 0x100659600
  Rht: 0x100659650
  Anc: 0x100603a70
  Name: ""
  BrLen: 0.448062
  Time: 0.954828
Node 1 (0x100659600)
  Lft: 0x0
  Rht: 0x0
  Anc: 0x100659570
  Name: "2"
  BrLen: 0.0451723
  Time: 1
Node 2 (0x100659650)
  Lft: 0x0
  Rht: 0x0
  Anc: 0x100659570
  Name: "3"
  BrLen: 0.0451723
  Time: 1
```

(You probably noticed that I modified the `print` function in the `Node` class to print the time for the node.) Still not pretty output. But, consider this: We made a birth-death tree!

9.3 Printing the tree as a Newick string

The Newick tree format was designed and adopted by several of the early developers of phylogeny software packages — J. Felsenstein, D. Swofford, F. J. Rohlf, C. Meacham, J. Archie, and W. Maddison — at Newick’s restaurant in Durhan, New Hampshire, on June 24, 1986. They wanted a standardized format describing trees so that any of their programs could read trees produced by another program. The meeting at the restaurant was motivated by conversations at the 1986 Evolution Meetings (the joint meetings of the Society of the Systematic Biologists, the American Society of Naturalists, and the Society for the Study of Evolution). They hunkered down to work at Newick’s restaurant while the meeting was underway. The Newick tree format was the result. According to D. Swofford, the meal at Newick’s restaurant was a good one. Perhaps he should write a long-overdue Yelp review?

The Newick format represents trees using parentheses. For example, the tree of Figure 8.1a can be represented as

((Sp1, Sp2), Sp3)

This format has several advantages: a single tree can be written to a plain text file on a single line; trees can be directly created in computer memory by reading the Newick string from left to right; and it requires few character to represent a tree.

Add to your **Tree** class two functions:

Example

```
std::string Tree::getNewickRepresentation(void) {

    std::stringstream ss;
    writeTree(root->getLft(), ss);
    std::string newick = ss.str();
    return newick;
}

void Tree::writeTree(Node* p, std::stringstream& ss) {

    if (p != NULL)
    {
        if (p->getLft() == NULL)
        {
            ss << p->getName() << ":" << std::fixed << std::setprecision(5) <<
                p->getBranchLength();
        }
        else
        {
```

```

        ss << "(";
        writeTree (p->getLft(), ss);
        ss << ",";
        writeTree (p->getRht(), ss);
        if (p->getAnc() == NULL)
            ss << ")";
        else
            ss << "):" << std::fixed << std::setprecision(5) << p->getBranchLength();
        }
    }
}

```

Your complete header file (`Tree.hpp`) should now look like:

Example

```

#ifndef Tree_hpp
#define Tree_hpp

#include <set>
#include <sstream>
#include <string>
#include <vector>
class Node;
class RandomVariable;

class Tree {

public:
    Tree(double lambda, double mu, double duration,
        RandomVariable* rv);
    ~Tree(void);
    std::vector<Node*>& getTraversalOrder(void) { return postOrderSequence; }
    void listNodes(void);
    std::string getNewickRepresentation(void);
    int getNumExtant(void) { return numExtant; }
}

```

```

        void                setNumExtant(int x) { numExtant = x; }

protected:
        Tree(void) { }
        Node*             chooseNodeFromSet(std::set<Node*>& s, RandomVariable* rv);
        void              initializeTraversalOrder(void);
        void              passDown(Node* p);
        void              writeTree(Node* p, std::stringstream& ss);
        Node*             root;
        std::vector<Node*> nodes;
        std::vector<Node*> postOrderSequence;
        int               numExtant;
};

#endif

```

I made the `getNewickRepresentation` function public but the `writeTree` function protected. Why do you think that is? Because the `writeTree` function uses the string stream, we need to include `sstream` in the `Tree.hpp` file (where it is first used).

It is through the `getNewickRepresentation` function that other parts of the program can obtain a string with the Newick representation of the tree. This function returns a `string`. The code for that function,

```

std::stringstream ss;
writeTree(root->getLft(), ss);
std::string newick = ss.str();
return newick;

```

first declares a `stringstream` variable, called `ss`. A `stringstream` operates much like the `iostream` that you have been using to direct characters to the console using `std::cout`. But, `sstream` acts much like a string, too. We use the functionality of this object in the `writeTree` function, to output the birth-death tree in Newick format. Note that after inserting characters into the string stream in `writeTree`, we convert it to a `string` object that is then returned from the `getNewickRepresentation` function.

Like the `passDown` function we used in the previous chapter to initialize the postorder traversal sequence for the tree, the `writeTree` function is also recursive. However, in this function, we do things before, between, and after the two recursive function calls.

Modify the `main` function so that it reads:

Example

```

#include <iostream>
#include "RandomVariable.hpp"
#include "Tree.hpp"

int main(int argc, char* argv[]) {

    RandomVariable rv;

    Tree t(3.0, 1.0, 1.0, &rv);
    std::cout << t.getNewickRepresentation() << std::endl;

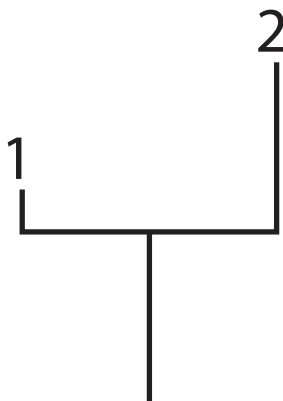
    return 0;
}

```

Compile and run the code. I ran the program, obtaining the following output:

```
(1:0.04425,2:0.14025):0.14240
```

which represents a tree of two species:



This is a tree that went extinct before reaching the end of the simulation (specified by the **duration** parameter). Because we generated this tree with the parameters $\lambda = 3$, $\mu = 1$, and $t = 1$, we know that the probability of the tree going extinct is approximately 0.3.

Several programs will display Newick strings as nicely-formatted trees. My favorite program for doing this is FigTree (<http://tree.bio.ed.ac.uk/software/figtree/>), which was written by A. Rambaut. Run the FigTree program, if you happen to have it. Generate a tree under the birth-death process of cladogenesis using the program you just wrote. Copy the Newick string

that is output by your program and paste it into the open FigTree window. You should see the representation of your tree in the window. It should look pretty!

Figure 9.2 shows an example of the output of the program when I ran it on my computer. You can see that this tree has 38 extant species and 7 extinct lineages. This is a view of the process that evolutionary biologists are never afforded. The fossil record, of course, gives a glimpse into which species existed in the past. If a group does not have any fossils, the only information available is through the living taxa. The reconstructed process prunes away any lineages that did not happen to make it to the present. This would involve pruning away the 7 extinct lineages. Remarkably, inferences about the speciation and extinction rates can be made on the reconstructed tree (Nee et al., 1994).

Our implementation of the birth-death process of cladogenesis has no bells and whistles. Rather, it implements the simple birth-death process. Evolutionary biologists are now interested in estimating parameters of embellished versions of the birth-death process. For example, we think the speciation and extinction rates can change over the course of time and across different lineages. Certainly, mass extinction events represent times when the extinction rate is higher. How would you modify the program to simulate mass extinction? How would you modify the program to allow the rates of speciation and extinction to change on the tree?

You will undoubtedly start playing with this program now that it has been successfully implemented. The program has several parameter that you can change, and I don't doubt that you will immediately start changing the speciation and extinction rates (`lambda` and `mu`). Go for it! But, beware: birth-death trees can become quite large very quickly. For our combination of parameters

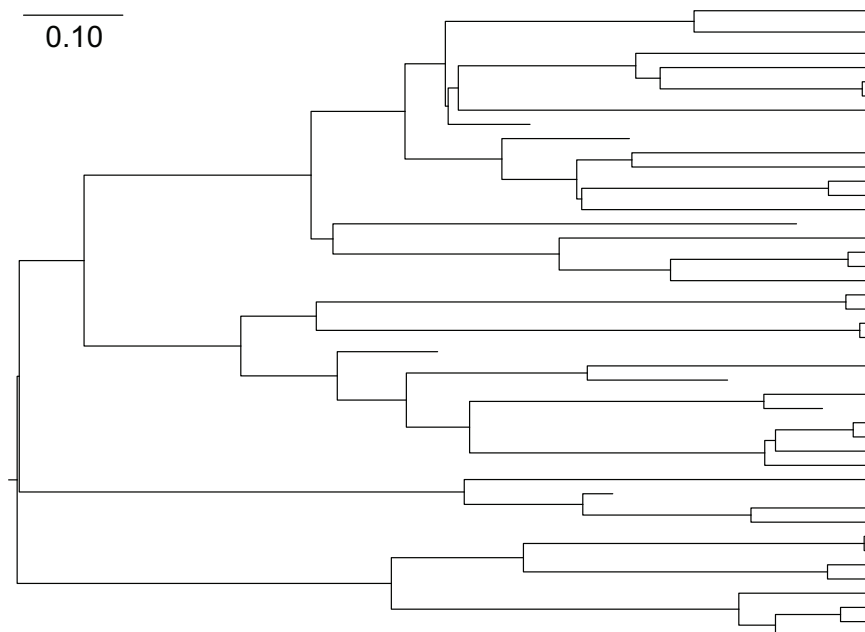


Figure 9.2: A single realization of the birth-death process of cladogenesis generated by the program written in this chapter. The tree was displayed using FigTree.

— $\lambda = 3$, $\mu = 1$, and $t = 1$ — the expected number of species is $E(N) = 7.40$. (Remember, the expected number of extant species is $E(N) = e^{(\lambda-\mu)t}$.) If we were to change λ to $\lambda = 10$, we would, on average, expect to see $E(N) = 8,103.08$ tips! This is a number that could be handled by the program, but I'm not certain it could handle a change in λ to $\lambda = 20$, in which case $E(N) = 178,482,300.96$. One possible change you could make to the code is to set a maximum number of nodes in the tree. If the tree attempts to grow past this maximum, you could exit the program with an error warning.

Chapter 10

Simulating Sequence Evolution on a Tree

10.1 Assumptions of phylogenetic methods

The models used in phylogenetic analysis of molecular data have three components. First, they assume a tree relating the samples. Here, the samples might be DNA sequences collected from different species, or different individuals within a population. In either case, a basic assumption is that the samples are related to one another through an (unknown) tree. This would be a species tree for sequences sampled from different species, or perhaps a coalescence tree for sequences sampled from individuals from within a population. Second, they assume that the branches of the tree have an (unknown) length. Ideally, the length of a branch on a tree is in terms of time. However, in practice it is difficult to determine the duration of a branch on a tree in terms of time. Instead, the lengths of the branches on the tree are in terms of expected change per character. Figure 10.1 shows some examples of trees with branch lengths. The main points the reader should remember are: (1) Trees can be rooted or unrooted. Rooted trees have a time direction whereas unrooted trees do not. Most methods of phylogenetic inference, including most implementations of maximum likelihood and Bayesian analysis, are based on time-reversible models of evolution that produce unrooted trees, which must be rooted using some other criterion, such as the outgroup criterion (using distantly related reference sequences to locate the root). (2) The space of possible trees is huge. The number of possible unrooted trees for n species is $B(n) = \frac{(2n-5)!}{2^{n-3}(n-3)!}$ (Schröder, 1870). This means that for a relatively small problem of only $n = 50$ species, there are about $B(50) = 2.838 \times 10^{74}$ possible unrooted trees that can explain the phylogenetic relationships of the species.

The third component of a phylogenetic model is a process that describes how the characters change on the phylogeny. All model-based methods of phylogenetic inference, including maximum likelihood and Bayesian estimation of phylogeny, currently assume that character change occurs according to a continuous-time Markov chain. At the heart of any continuous-time Markov chain is a matrix of rates, specifying the rate of change from one state to another. For example, the instantaneous rate of change under the model described by Hasegawa et al. (1984, 1985, hereafter

called the HKY85 model) is

$$\mathbf{Q} = \{q_{ij}\} = \begin{pmatrix} - & \pi_C & \kappa\pi_G & \pi_T \\ \pi_A & - & \pi_G & \kappa\pi_T \\ \kappa\pi_A & \pi_C & - & \pi_T \\ \pi_A & \kappa\pi_C & \pi_G & - \end{pmatrix} \mu$$

This matrix specifies the rate of change from one nucleotide to another; the rows and columns of the matrix are ordered A, C, G, T , so that the rate of change from $C \rightarrow G$ is $q_{CG} = \pi_G$. Similarly, the rates of change between $C \rightarrow T$, $G \rightarrow A$, and $T \rightarrow C$, are $q_{CT} = \kappa\pi_T$, $q_{GA} = \kappa\pi_A$, and $q_{TG} = \pi_G$, respectively. The diagonals of the rate matrix, denoted with the dashes, are specified such that each row sums to zero. Finally, the rate matrix is rescaled such that the mean rate of substitution is one. This can be accomplished by setting $\mu = -1/\sum_{i \in \{A,C,G,T\}} \pi_i q_{ii}$. This rescaling of the rate matrix such that the mean rate is one allows the branch lengths on the phylogenetic tree to be interpreted as expected number of nucleotide substitutions per site.

We will make a few important points about the rate matrix. First, the rate matrix may have free parameters. For example, the HKY85 model has the parameters κ , π_A , π_C , π_G , and π_T .

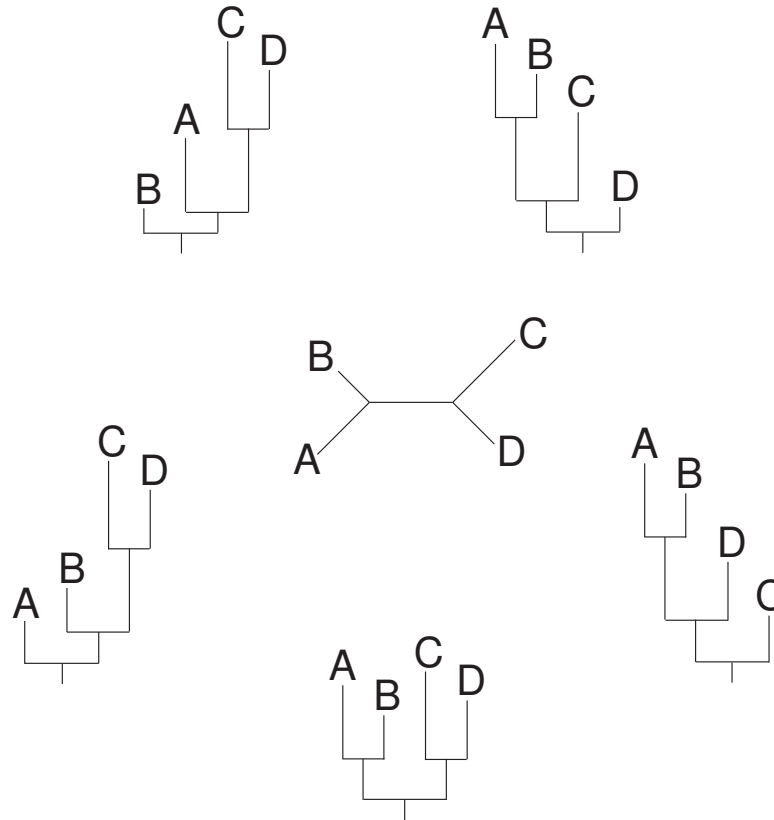


Figure 10.1: An unrooted tree of four species (center) with the branch lengths drawn proportional to their length in terms of expected number of substitutions per site. The five trees surrounding the central, unrooted, tree show the five possible rooted trees that result from the unrooted tree.

The parameter κ is the transition/transversion rate bias; when $\kappa = 1$ transitions occur at the same rate as transversions. Typically, the transition/transversion rate ratio, estimated using maximum likelihood or Bayesian inference, is greater than one; transitions occur at a higher rate than transversions. The other parameters — π_A , π_C , π_G , and π_T — are the base frequencies, and have a biological interpretation as the frequency of the different nucleotides and are also, incidentally, the stationary probabilities of the process (more on stationary probabilities later). Second, the rate matrix, \mathbf{Q} , can be used to calculate the transition probabilities and the stationary distribution of the substitution process. The transition probabilities and stationary distribution play a key role in calculating the likelihood, and we will spend more time here developing an intuitive understanding of these concepts.

10.1.1 Transition probabilities

Let us consider a specific example of a rate matrix, with all of the parameters of the model taking specific values. For example, if we use the HKY85 model and fix the parameters to $\kappa = 5$, $\pi_A = 0.4$, $\pi_C = 0.3$, $\pi_G = 0.2$, and $\pi_T = 0.1$, we get the following matrix of instantaneous rates

$$\mathbf{Q} = \{q_{ij}\} = \begin{pmatrix} -0.886 & 0.190 & 0.633 & 0.063 \\ 0.253 & -0.696 & 0.127 & 0.316 \\ 1.266 & 0.190 & -1.519 & 0.063 \\ 0.253 & 0.949 & 0.127 & -1.329 \end{pmatrix}$$

Note that these numbers are not special in any particular way. That is to say, they are not based upon any observations from a real data set, but are rather arbitrarily picked to illustrate a point. The point is that one can interpret the rate matrix in the physical sense of specifying how changes occur on a phylogenetic tree. Consider the very simple case of a single branch on a phylogenetic tree. Let's assume that the branch is $v = 0.5$ in length and that the ancestor of the branch is the nucleotide G . The situation we have is something like that shown in Figure 10.2a. How can we simulate the evolution of the site starting from the G at the ancestor? The rate matrix tells us how to do this. First of all, because the current state of the process is G , the only relevant row of the rate matrix is the third one:

$$\mathbf{Q} = \{q_{ij}\} = \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ 1.266 & 0.190 & -1.519 & 0.063 \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

The overall rate of change away from nucleotide G is $q_{GA} + q_{GC} + q_{GT} = 1.266 + 0.190 + 0.063 = 1.519$. Equivalently, the rate of change away from nucleotide G is simply $-q_{GG} = 1.519$. In a continuous-time Markov model, the waiting time between substitutions is exponentially distributed. The exact shape of the exponential distribution is determined by its rate, which is the same as the rate of the corresponding process in the \mathbf{Q} matrix. For instance, if we are in state G , we wait an exponentially distributed amount of time with rate 1.519 until the next substitution occurs. One can easily construct exponential random variables from uniform random variables using the equation

$$t = -\frac{1}{\lambda} \log_e(u)$$

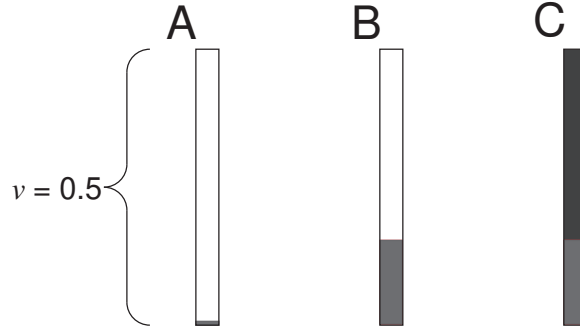


Figure 10.2: A single realization of the substitution process under the HKY85 model when $\kappa = 5$, $\pi_A = 0.4$, $\pi_C = 0.3$, $\pi_G = 0.2$, and $\pi_T = 0.1$. The length of the branch is $v = 0.5$ and the starting nucleotide is G (light gray). A, The process starts in nucleotide G . B, The first change is 0.152 units up the branch. C, the change is from G to A (dark gray). The time at which the next change occurs exceeds the total branch length, so the process ends in state C .

where λ is the rate and u is a uniform(0,1) random number. For example, my calculator has a uniform(0,1) random number generator. The first number it generated is $u = 0.794$. This means that the next time at which a substitution occurs is 0.152 up from the root of the tree (using $\lambda = 1.519$; Figure 10.2b). The rate matrix also specifies the probabilities of a change from G to the nucleotides A , C , and T . These probabilities are

$$G \rightarrow A : \frac{1.266}{1.519} = 0.833, \quad G \rightarrow C : \frac{0.190}{1.519} = 0.125, \quad G \rightarrow T : \frac{0.063}{1.519} = 0.042$$

To determine what nucleotide the process changes to we would generate another uniform(0,1) random number (again called u). If u is between 0 and 0.833, we will say that we had a change from G to A . If the random number is between 0.833 and 0.958 we will say that we had a change from G to C . Finally, if the random number u is between 0.958 and 1.000, we will say we had a change from G to T . The next number generated on our calculator was $u = 0.102$, which means the change was from G to A . The process is now in a different state (the nucleotide A) and the relevant row of the rate matrix is

$$\mathbf{Q} = \{q_{ij}\} = \begin{pmatrix} -0.886 & 0.190 & 0.633 & 0.063 \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

We wait an exponentially distributed amount of time with parameter $\lambda = 0.886$ until the next substitution occurs. When the substitution occurs, it is to a C , G , or T with probabilities $\frac{0.190}{0.886} = 0.214$, $\frac{0.633}{0.886} = 0.714$, and $\frac{0.063}{0.886} = 0.072$, respectively. This process of generating random and exponentially distributed times until the next substitution occurs and then determining (randomly) what nucleotide the change is to is repeated until the process exceeds the length of the branch. The state the process is in when it passes the end of the branch is recorded. In the example of Figure 10.2, the process started in state G and ended in state A . (The next uniform random variable generated on our calculator was $u = 0.371$, which means that the next substitution would

occur 1.119 units above the substitution from $G \rightarrow A$. The process is in the state A when it passed the end of the branch.) The only non-random part of the entire procedure was the initial decision to start the process in state G . All other aspects of the simulation used a uniform random number generator and our knowledge of the rate matrix to simulate a single realization of the HKY85 process of DNA substitution.

This Monte Carlo procedure for simulating the HKY85 process of DNA substitution can be repeated. The following table summarizes the results of 100 simulations, each of which started with the nucleotide G :

Starting Nucleotide	Ending Nucleotide	Number of Replicates
G	A	27
G	C	10
G	G	59
G	T	4

This table can be interpreted as a Monte Carlo approximation of the *transition probabilities* from nucleotide G to nucleotide $i \in (A, C, G, T)$. Specifically, the Monte Carlo approximations are $p_{GA}(0.5) \approx 0.27$, $p_{GC}(0.5) \approx 0.10$, $p_{GG}(0.5) \approx 0.59$, and $p_{GT}(0.5) \approx 0.04$. These approximate probabilities are all conditioned on the starting nucleotide being G and the branch length being $v = 0.5$. We performed additional simulations in which the starting nucleotide was A , C , or T . Together with the earlier Monte Carlo simulation that started with the nucleotide G , these additional simulations allow us to fill out the following table with the approximate transition probabilities:

		Ending Nucleotide			
		A	C	G	T
Starting Nucleotide	A	0.67	0.13	0.20	0.00
	C	0.13	0.70	0.07	0.10
	G	0.27	0.10	0.59	0.04
	T	0.12	0.30	0.08	0.50

Clearly, these numbers are only crude approximations to the true transition probabilities; after all, each row in the table is based on only 100 Monte Carlo simulations. However, they do illustrate the meaning of the transition probabilities; the transition probability, $p_{ij}(v)$, is the probability that the substitution process ends in nucleotide j conditioned on it starting in nucleotide i after an evolutionary amount of time v . The table of approximate transition probabilities, above, can be interpreted as a matrix of probabilities, usually denoted $\mathbf{P}(v)$. Fortunately, we do not need to rely on Monte Carlo simulation to approximate the transition probability matrix. Instead, we can calculate the transition probability matrix exactly using matrix exponentiation:

$$\mathbf{P}(v) = e^{\mathbf{Q}v}$$

For the case we have been simulating, the exact transition probabilities (to four decimal places) are

$$\mathbf{P}(0.5) = \{p_{ij}(0.5)\} = \begin{pmatrix} 0.7079 & 0.0813 & 0.1835 & 0.0271 \\ 0.1085 & 0.7377 & 0.0542 & 0.0995 \\ 0.3670 & 0.0813 & 0.5244 & 0.0271 \\ 0.1085 & 0.2985 & 0.0542 & 0.5387 \end{pmatrix}$$

The transition probability matrix accounts for all the possible ways the process could end up in nucleotide j after starting in nucleotide i . In fact, each of the infinite possibilities is weighted by its probability under the substitution model.

10.1.2 Stationary distribution

The transition probabilities provide the probability of ending in a particular nucleotide after some specific amount of time (or opportunity for substitution, v). These transition probabilities are conditioned on starting in a particular nucleotide. What do the transition probability matrices look like as v increases? The following transition probability matrices show the effect of increasing branch length:

$$\begin{aligned}
 \mathbf{P}(0.00) &= \begin{pmatrix} 1.000 & 0.000 & 0.000 & 0.000 \\ 0.000 & 1.000 & 0.000 & 0.000 \\ 0.000 & 0.000 & 1.000 & 0.000 \\ 0.000 & 0.000 & 0.000 & 1.000 \end{pmatrix} & \mathbf{P}(0.01) &= \begin{pmatrix} 0.991 & 0.002 & 0.006 & 0.001 \\ 0.003 & 0.993 & 0.001 & 0.003 \\ 0.013 & 0.002 & 0.985 & 0.001 \\ 0.003 & 0.009 & 0.001 & 0.987 \end{pmatrix} \\
 \mathbf{P}(0.10) &= \begin{pmatrix} 0.919 & 0.018 & 0.056 & 0.006 \\ 0.024 & 0.934 & 0.012 & 0.029 \\ 0.113 & 0.018 & 0.863 & 0.006 \\ 0.025 & 0.086 & 0.012 & 0.877 \end{pmatrix} & \mathbf{P}(0.50) &= \begin{pmatrix} 0.708 & 0.081 & 0.184 & 0.027 \\ 0.106 & 0.738 & 0.054 & 0.100 \\ 0.367 & 0.081 & 0.524 & 0.027 \\ 0.109 & 0.299 & 0.054 & 0.539 \end{pmatrix} \\
 \mathbf{P}(1.00) &= \begin{pmatrix} 0.580 & 0.141 & 0.232 & 0.047 \\ 0.188 & 0.587 & 0.094 & 0.131 \\ 0.464 & 0.141 & 0.348 & 0.047 \\ 0.188 & 0.394 & 0.094 & 0.324 \end{pmatrix} & \mathbf{P}(5.00) &= \begin{pmatrix} 0.411 & 0.287 & 0.206 & 0.096 \\ 0.383 & 0.319 & 0.192 & 0.106 \\ 0.411 & 0.287 & 0.206 & 0.096 \\ 0.383 & 0.319 & 0.192 & 0.107 \end{pmatrix} \\
 \mathbf{P}(10.0) &= \begin{pmatrix} 0.401 & 0.299 & 0.200 & 0.099 \\ 0.399 & 0.301 & 0.199 & 0.100 \\ 0.401 & 0.299 & 0.200 & 0.099 \\ 0.399 & 0.301 & 0.199 & 0.100 \end{pmatrix} & \mathbf{P}(100) &= \begin{pmatrix} 0.400 & 0.300 & 0.200 & 0.100 \\ 0.400 & 0.300 & 0.200 & 0.100 \\ 0.400 & 0.300 & 0.200 & 0.100 \\ 0.400 & 0.300 & 0.200 & 0.100 \end{pmatrix}
 \end{aligned}$$

(Each matrix was calculated under the HKY85 model with $\kappa = 5$, $\pi_A = 0.4$, $\pi_C = 0.3$, $\pi_G = 0.2$, and $\pi_T = 0.1$.) Note that as the length of a branch, v , increases, the probability of ending up in a particular nucleotide converges to a single number, regardless of the starting state. For example, the probability of ending up in C is about 0.300 when the branch length is $v = 100$. This is true regardless of whether the process starts in A , C , G , or T . The substitution process has in a sense ‘forgotten’ its starting state.

The stationary distribution is the probability of observing a particular state when the branch length increases without limit ($v \rightarrow \infty$). The stationary probabilities of the four nucleotides are $\pi_A = 0.4$, $\pi_C = 0.3$, $\pi_G = 0.2$, and $\pi_T = 0.1$ for the example discussed above. The models typically used in phylogenetic analyses have the stationary probabilities built into the rate matrix, \mathbf{Q} . You will notice that the rate matrix for the HKY85 model has parameters π_A , π_C , π_G , and π_T , and that the stationary frequencies of the four nucleotides for our example match the input values for our simulations. Building the stationary frequency of the process into the rate matrix, while somewhat unusual, makes calculating the likelihood function easier. For one, specifying the stationary distribution saves the time of figuring out what the stationary distribution is (which

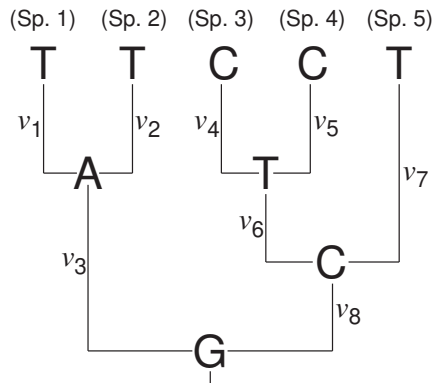


Figure 10.3: One of the possible (rooted) trees describing the evolutionary history of the five species. The states at the first site in the alignment of the text are shown at the tips of the tree. The states at the interior nodes of the tree are also shown, though in reality these states are not observed. The length of the i th branch is denoted v_i .

involves solving the equation $\pi\mathbf{Q} = \mathbf{0}$, which simply says that, if we start with the nucleotide frequencies reflecting the stationary distribution, the process will have no effect on the nucleotide frequencies). For another, it allows one to more easily specify a time reversible substitution model. [A time reversible substitution model has the property that $\pi_i q_{ij} = \pi_j q_{ji}$ for all $i, j \in (A, C, G, T)$, $i \neq j$.] Practically speaking, time reversibility means that we can work with unrooted trees instead of rooted trees (assuming that the molecular clock is not enforced).

Calculating the likelihood

The transition probabilities and stationary distribution are used when calculating the likelihood. For example, consider the following alignment of sequences for five species¹:

Species 1	TAACTGTAAAGGACAACACTAGCAGGCCAGACGCACACGCACAGCGCACC
Species 2	TGACTTTAAAGGACGACCCTACCAGGGCGGACACAAACGGACAGCGCAGC
Species 3	CAAGTTTAGAAAACGGCACCAACACAACAGACGTATGCAACTGACGCACC
Species 4	CGAGTTCAGAAGACGGCACCAACACAGCGGACGTATGCAGACGACGCACC
Species 5	TGCCCTTAGGAGGCGGCACTAACACCGCGGACGAGTGCGGACAACGTACC

This is clearly a rather small alignment of sequences to use for estimating phylogeny, but it will illustrate how likelihoods are calculated. The likelihood is the probability of the alignment of sequences, conditioned on a tree with branch lengths. The basic procedure is to calculate the probability of each site (column) in the matrix. Assuming that the substitutions are independent across sites, the probability of the entire alignment is simply the product of the probabilities of the individual sites.

How is the likelihood at a single site calculated? Figure 10.3 shows the observations at the first site (T , T , C , C , and T) at the tips of one of the possible phylogenetic trees for five species. The

¹This alignment was simulated on the tree of Figure 10.3 under the HKY85 model of DNA substitution. Parameter values for the simulation can be found in the caption of Table 1.

tree in Figure 10.3 is unusual in that we will assume that the nucleotide states at the interior nodes of the tree are also known. This is clearly a bad assumption, because we cannot directly observe the nucleotides that occurred at any point on the tree in the distant past. For now, however, ignore this fact and bear with me. The probability of observing the configuration of nucleotides at the tips and interior nodes of the tree in Figure 10.3 is

$$\mathbb{P}(TTCCT, ATCG|\tau, \mathbf{v}, \theta) = \pi_G p_{GA}(v_3) p_{AT}(v_1) p_{AT}(v_2) p_{GC}(v_8) p_{CT}(v_6) p_{CT}(v_7) p_{TC}(v_4) p_{TC}(v_5)$$

Here we show the probability of the observations (TTCCT) and the states at the interior nodes of the tree (ATCG) conditioned on the tree (τ), branch lengths (\mathbf{v}), and other model parameters (θ). Note that to calculate the probability of the states at the tips of the tree, we used the stationary probability of the process (π) and also the transition probabilities $[p_{ij}(v)]$. The stationary probability of the substitution process was used to calculate the probability of the nucleotide at the root of the tree. In this case, we are assuming that the substitution process has been running a very long time before it reached the root of our five species tree. We then use the transition probabilities to calculate the probabilities of observing the states at each end of the branches. When taking the product of the transition probabilities, we are making the additional assumption that the substitutions on each branch of the tree are independent of one another. This is probably a reasonable assumption for real data sets.

The probability of observing the states at the tips of the tree, described above, was conditioned on the interior nodes of the tree taking specific values (in this case *ATCG*). To calculate the unconditional probability of the observed states at the tips of the tree, we sum over all possible combinations of nucleotide states that can be assigned to the interior nodes of the tree

$$\mathbb{P}(TTCCT|\tau, \mathbf{v}, \theta) = \sum_w \sum_x \sum_y \sum_z \mathbb{P}(TTCCT, wxyz|\tau, \mathbf{v}, \theta)$$

where $w, x, y, z \in (A, C, G, T)$. Averaging the probabilities over all combinations of states at the interior nodes of the tree accomplishes two things. First, we remove the assumption that the states at the interior nodes take specific values. Second, because the transition probabilities account for all of the possible ways we could have state i at one end of a branch and state j at the other, the probability of the site is also averaged over all possible character histories. Here, we think of a character history as one realization of changes on the tree that is consistent with the observations at the tips of the tree. For example, the parsimony method, besides calculating the minimum number of changes on the tree, also provides a character history; the character history favored by parsimony is the one that minimizes the number of changes required to explain the data. In the case of likelihood-based methods, the likelihood accounts for all possible character histories, with each history weighted by its probability under the substitution model. (Nielsen, 2002) described a method for sampling character histories in proportion to their probability that relies on the interpretation of the rate matrix as specifying waiting times between substitutions. His method provides a means to reconstruct the history of a character that does not inherit the flaws of the parsimony method. Namely, Nielsen's method allows multiple changes on a single branch and also allows for non-parsimonious reconstructions of a character's history.

Before moving on, we will make two final points. First, in practice no computer program actually evaluates all combinations of nucleotides that can be assigned to the interior nodes of a tree when

Site	Prob.	Site	Prob.	Site	Prob.	Site	Prob.	Site	Prob.
1	0.004025	11	0.029483	21	0.179392	31	0.179392	41	0.003755
2	0.001171	12	0.006853	22	0.001003	32	0.154924	42	0.005373
3	0.008008	13	0.024885	23	0.154924	33	0.007647	43	0.016449
4	0.002041	14	0.154924	24	0.179392	34	0.000936	44	0.029483
5	0.005885	15	0.007647	25	0.005719	35	0.024885	45	0.154924
6	0.000397	16	0.024124	26	0.001676	36	0.000403	46	0.047678
7	0.002802	17	0.154924	27	0.000161	37	0.024124	47	0.010442
8	0.179392	18	0.004000	28	0.154924	38	0.154924	48	0.179392
9	0.024124	19	0.154924	29	0.001171	39	0.011088	49	0.002186
10	0.024885	20	0.004025	30	0.047678	40	0.000161	50	0.154924

Table 10.1: The probabilities of the fifty sites for the example alignment from the text. The likelihoods are calculated assuming the tree of Figure 10.3 with the branch lengths being $v_1 = 0.1$, $v_2 = 0.1$, $v_3 = 0.2$, $v_4 = 0.1$, $v_5 = 0.1$, $v_6 = 0.1$, $v_7 = 0.2$, and $v_8 = 0.1$. The substitution model parameters were also fixed, with $\kappa = 5$, $\pi_A = 0.4$, $\pi_C = 0.3$, $\pi_G = 0.2$, and $\pi_T = 0.1$.

calculating the probability of observing the data at a site. There are simply too many combinations for trees of even small size. For example, for a tree of 100 species, there are 99 interior nodes and 4.02×10^{59} combinations of nucleotides at the ancestral nodes on the tree. Instead, the Felsenstein (1981) pruning algorithm is used to calculate the likelihood at a site. Felsenstein's method is mathematically equivalent to the summation shown above, but can evaluate the likelihood at a site in a fraction of the time it would take to plow through all combinations of ancestral states. Second, the overall likelihood of a character matrix is the product of the site likelihoods. If we assume that the tree of Figure 10.3 is correct (with all of the parameters taking the values specified in the caption of Table 10.1.2), then the probability of observing the data is

$$0.004025 \times 0.001171 \times 0.008008 \times \dots \times 0.154924 = 1.2316 \times 10^{-94}$$

where there are fifty factors, each factor representing the probability of an individual site (column) in the alignment. Table 10.1.2 shows the probabilities of all fifty sites for the tree of Figure 10.3. Note that the overall probability of observing the data is a very small number ($\approx 10^{-94}$). This is typical of phylogenetic problems and results from the simple fact that many numbers between 0 and 1 are multiplied together. Computers cannot accurately hold very small numbers in memory. Programmers avoid this problem of computer “underflow” by using the log probability of observing the data. The log probability of observing the sample alignment of sequences presented earlier is $\log_e \ell = \log_e(1.2316 \times 10^{-94}) = -216.234734$. The log likelihood can be accurately stored in computer memory.

10.2 Four equivalent ways to simulate DNA sequences on a tree

In the previous section, we covered some of the most basic assumptions made in a phylogenetic analysis; DNA sequences are assumed to evolve on a phylogenetic tree (with branch lengths) under a continuous-time Markov model of DNA substitution. The substitution process is assumed to be independent across sites. These assumptions apply to maximum likelihood, Bayesian inference,

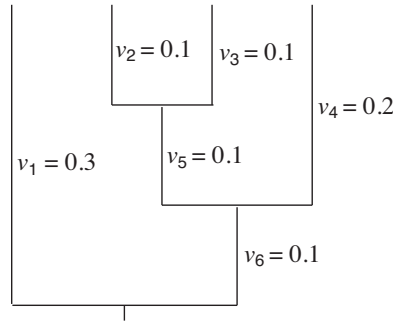


Figure 10.4: We simulate data on this tree. The branch lengths are denoted v_i .

and distance methods (when the distances are ‘corrected’ under some evolutionary model). It is not so clear what assumptions the parsimony method is making (one potential criticism of the method), but several authors have found interesting connections between the parsimony method and maximum likelihood under an over-parameterized continuous-time Markov model of DNA substitution (Tuffley and Steel, 1997). .

In this section, we will test our knowledge obtained in the previous section by simulating DNA sequences on a phylogenetic tree. Simulation is often used in phylogenetics. Simulation has been used to elucidate the statistical properties of different phylogenetic methods, and it can be used to generate the null distribution of a test statistic in phylogenetic hypothesis testing. In other words, learning to simulate DNA sequences is not a wasted effort. Not only can you strengthen your intuition of phylogenetic methods, but you may also be able to apply simulation for your own research.

How exactly should one simulate evolution on a phylogenetic tree? One basic point is that an alignment should be simulated on a site-by-site basis. That is, we first simulate the data at the first site, then the second site, and so on. We can take this approach because of the assumption of independence of the substitution process across sites; to simulate the data at a particular site (column in the alignment), we don’t need to know the results of the simulation at any other site.

Another basic point is that we must know all of the parameters of the simulation: we have to decide on the precise phylogenetic tree on which to simulate the DNA sequences; we need to know the branch lengths on this tree; and, finally, we must pick a substitution model. The substitution model is a matrix of rates, specifying the rate of change from one nucleotide to another. In other words, we are taking a God-like view of the situation. We know *everything* about the evolutionary history and process. Of course, in reality we never know everything about how organisms evolved, but must make strong assumptions about how evolution occurred in order to estimate (make educated guesses) at the underlying evolutionary history. However, pretending to be a God, even for a little while, is a great feeling.

In the following, we will evolve DNA sequences on the four-taxon tree shown in Figure 10.4. We will also assume that DNA substitution occurs according to the HKY85 model with the parameters fixed to the following values: $\kappa = 5$, $\pi_A = 0.4$, $\pi_C = 0.3$, $\pi_G = 0.2$, and $\pi_T = 0.1$. The rate matrix,

then, is

$$\mathbf{Q} = \{q_{ij}\} = \begin{pmatrix} -0.886 & 0.190 & 0.633 & 0.063 \\ 0.253 & -0.696 & 0.127 & 0.316 \\ 1.266 & 0.190 & -1.519 & 0.063 \\ 0.253 & 0.949 & 0.127 & -1.329 \end{pmatrix}$$

Now, we are ready to simulate data on the tree of Figure 10.4. We will go over four different methods for simulating data, each of which takes advantage of our knowledge of continuous-time Markov chains.

10.2.1 Method 1

The first method only relies on our ability to generate exponentially distributed random numbers. If we generate a uniform random number on the interval (0,1), we can generate an exponential random number (with parameters λ) using the transformation $t = -\frac{1}{\lambda} \log_e(u)$ (where u is the uniform random number and t is the exponentially distributed random number). The first method involves an addition to the tree of Figure 10.4 which seems unusual: We take the tree of Figure 10.4, and add a ‘tail’ to it—a branch that extends for some distance from the root of the tree. In this case, the branch at the root of the tree is $v_0 = 10.0$ in length. Moreover, we assume that the process is in state (nucleotide) A at the very root of the tree. The situation we have is like that shown in Figure 10.5.

We simulate the process starting at the root of the tree. The process is in state A, meaning that the only relevant row of the rate matrix is the first one:

$$\mathbf{Q} = \{q_{ij}\} = \begin{pmatrix} -0.886 & 0.190 & 0.633 & 0.063 \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix}$$

We wait an exponentially distributed amount of time with parameter $\lambda = 0.886$ until the next substitution occurs. When the substitution occurs, it is to a C, G, or T with probabilities $\frac{0.190}{0.886} = 0.214$, $\frac{0.633}{0.886} = 0.714$, and $\frac{0.063}{0.886} = 0.072$, respectively. In the first section, we used this method for simulating along a single branch of a tree. Here we apply the method with vigor, applying it to each branch in the tree from the root to the tips. We continue to simulate up the root branch of the tree until our simulation exceeds the length of the branch. We then record the nucleotide state the process was in when it exceeded a length of 10. We write this state at the end of the root branch, where it splits into branches 1 and 6. We then repeat the simulation process for branch 1 and then branch 6, recording the state the process is in at the end of those two branches. We then concentrate our attention on branches 4 and 5, and then on branches 2 and 3. At the end, we should have nucleotides at the ends of branches 1, 2, 3, and 4.

One puzzling aspect of this simulation is why we always start the process in nucleotide A, and why we even bothered to add the tail to the root of the tree. We did this because for Method 1, we assume ignorance of the stationary distribution of the rate matrix. If this is the case, we can use our understanding of the rate matrix as specifying waiting times between substitutions to complete our simulation. Hence, we always start our simulations in a particular nucleotide (in this case we chose to start in the nucleotide A), and then simulate the process for a long time along the root (tail) branch of the tree. The hope is that if we make the length of the tail branch long enough,

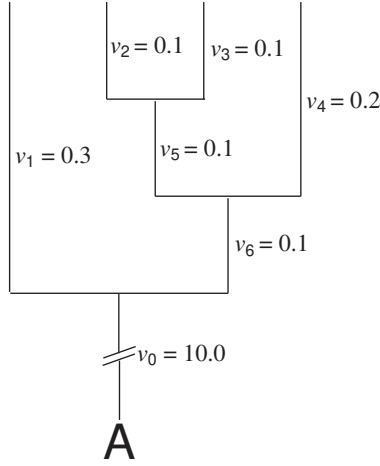


Figure 10.5: The tree of Figure 10.4, with a long branch at the root that starts in nucleotide A.

that the process is at stationarity by the time it reaches the first split in the tree (the speciation event that eventually produces the four species at the tips of the tree).

Method 1 relies on the idea that we can come pretty near to stationarity with a moderately long branch. We know that the stationary distribution of the HKY85 process of nucleotide substitution with the specific parameters we chose is $\pi_A = 0.4$, $\pi_C = 0.3$, $\pi_G = 0.2$, $\pi_T = 0.1$. We also know that the transition probability for a branch of $v = 10.0$ is

$$\mathbf{P}(10.0) = \begin{pmatrix} 0.401 & 0.299 & 0.200 & 0.099 \\ 0.399 & 0.301 & 0.199 & 0.100 \\ 0.401 & 0.299 & 0.200 & 0.099 \\ 0.399 & 0.301 & 0.199 & 0.100 \end{pmatrix}$$

The transition probability matrix tells us that if we start in nucleotide A, then we end up in state A, C, G, and T with probabilities 0.401, 0.299, 0.200, and 0.099, respectively. These numbers are very close to the actual stationary probabilities, so perhaps this method is not such a bad one.

10.2.2 Method 2

The second simulation method we explore is similar to the first one, but we eliminate the long tail on the tree. Instead of simulating the process along a long branch before we reach the first split in the tree, we simply decide which nucleotide is at the first split by sampling from the stationary distribution. We know that the stationary probabilities of the process are $\pi_A = 0.4$, $\pi_C = 0.3$, $\pi_G = 0.2$, $\pi_T = 0.1$, so why not simply pick a nucleotide at random with these probabilities?

10.2.3 Method 3

The third simulation method does away with the need to generate exponential random variables. It takes advantage of our knowledge of the stationary distribution (as does the second method), but also takes advantage of our ability to calculate transition probabilities. There are only three

different lengths of branches on our model tree (0.1, 0.2, and 0.3). The transition probabilities are

$$\mathbf{P}(0.10) = \begin{pmatrix} 0.919 & 0.018 & 0.056 & 0.006 \\ 0.024 & 0.934 & 0.012 & 0.029 \\ 0.113 & 0.018 & 0.863 & 0.006 \\ 0.025 & 0.086 & 0.012 & 0.877 \end{pmatrix} \quad \mathbf{P}(0.20) = \begin{pmatrix} 0.851 & 0.035 & 0.100 & 0.011 \\ 0.047 & 0.876 & 0.023 & 0.052 \\ 0.201 & 0.035 & 0.750 & 0.011 \\ 0.047 & 0.156 & 0.023 & 0.771 \end{pmatrix}$$

$$\mathbf{P}(0.30) = \begin{pmatrix} 0.795 & 0.051 & 0.135 & 0.017 \\ 0.069 & 0.824 & 0.034 & 0.071 \\ 0.270 & 0.051 & 0.659 & 0.017 \\ 0.069 & 0.214 & 0.034 & 0.681 \end{pmatrix}$$

Instead of drawing exponential random variables, and generating the process continuously across the entire tree, our simulation jumps from node to node on the tree. First, we generate the nucleotide at the root of the tree (the first split leading to all four taxa) by drawing from the stationary distribution. Then, we use the transition probabilities to simulate from one end to the other of each branch on the tree. We start from the root of the tree, and simulate up the tree to progressively higher branches until we have simulated a nucleotide at each tip of the tree.

10.2.4 Method 4

The last method we will discuss involves calculating the probability of each possible pattern of nucleotides that we could observe at the tips of the tree. There are four tips, so there are $4^4 = 256$ possible patterns of nucleotides we could observe. We use the formula for the likelihood, discussed in the first section, to calculate the probability (likelihood) of each pattern.

We could quickly simulate the data at one site (column) in an alignment by generating one uniform random number on the interval (0,1). We would first decide which pattern would result from a particular random number by tabulating 256 intervals. For example, we might decide that if the uniform random number is between 0 and 0.199465, that the pattern AAAA is the result; that if the uniform random number is between 0.199465 and 0.20365 that the pattern AAAC results; that if the uniform random number is between 0.20365 and 0.218361 that the pattern AAAG results; and so on. These intervals are calculated directly from the numbers in Appendix 1. (If you do not see how this was done, here is a hint: $0.199465 + 0.004185 = 0.20365$ and $0.199465 + 0.004185 + 0.014711 = 0.218361$.)

This method can work well when the number of tips on the tree is small, keeping the number of possible patterns manageable. However, when the number of tips gets to be about 8 or 9, the number of possible patterns becomes too large to make this method a reasonable one for simulating data.

10.3 A C++ simulator

In order to simulate DNA sequence evolution on a phylogenetic tree, we will need the abilities to hold a tree in computer memory and to generate pseudorandom numbers. Fortunately, we have implemented both in earlier chapters. In fact, we will use the code from the previous chapter, in which we generate a tree under the birth-death process of cladogenesis, as the starting point for simulating DNA sequences. Make certain that your birth-death project is open.

10.3.1 Modifying the Tree class

First, we will add some functionality to our `Tree` class. We will make two new functions: another constructor and a function. The new constructor will set up the tree using a string in Newick format. Add to the header file for the `Tree` class (`Tree.hpp`), the following public constructor:

Example

```
Tree(std::string newickStr);
```

and the following private function:

Example

```
std::vector<std::string> parseNewickString(std::string ns);
```

Now, we need to implement the new constructor and function. In the `Tree.cpp` file, add the following code:

Example

```
Tree::Tree(std::string newickStr) {  
  
    // break the string into its component parts  
    std::vector<std::string> tokens = parseNewickString(newickStr);  
  
}  
  
std::vector<std::string> Tree::parseNewickString(std::string ns) {  
  
    std::vector<std::string> tks;  
    for (int i=0; i<ns.size(); i++)  
    {  
        char c = ns[i];  
        if (c == '(' || c == ')') || c == ',' || c == ':' || c == ';')  

```

```

        {
            std::string tempStr;
            tempStr = c;
            tks.push_back(tempStr);
        }
    else
    {
        int j = i;
        std::string tempStr = "";
        while ( !(c == '(' || c == ')' || c == ',' || c == ':' || c == ';' ) )
        {
            tempStr += c;
            j++;
            c = ns[j];
        }
        i = j-1;
        tks.push_back(tempStr);
    }
}

# if defined(DEBUG_NEWICK_PARSER)
std::cout << "The Newick string, broken into its parts:" << std::endl;
for (int i=0; i<tk.size(); i++)
    std::cout << "    tks[" << i << "] = \"" << tks[i] << "\"" << std::endl;
# endif
return tks;
}

```

You will need to add,

Example

```
#define DEBUG_NEWICK_PARSER
```

near the top of your `Tree.cpp` file. I would add the `#define` compiler directive after the `#include` directives, but before the actual code in the file. Modify your `main` function so that it reads:

Example

```

#include <iostream>
#include <string>
#include "RandomVariable.hpp"
#include "Tree.hpp"

int main(int argc, char* argv[]) {

    RandomVariable rv;

    std::string myTree = "(Taxon_I:0.3,((Taxon_II:0.1,Taxon_III:0.1):0.1,
                          Taxon_IV:0.2):0.1)";

    Tree t(myTree);

    return 0;
}

```

We haven't completed our implementation of the constructor, but this is a good point to pause. Check that your code compiles. It should at this point.

What have we done? The basic idea is to construct a tree so that it reflects the topology and branch lengths depicted in Figure 10.4. We initialize a string in the `main` function, called `myTree` that reflects the information in that tree. Note that I have added taxon names; the names `Taxon_I`, `Taxon_II`, `Taxon_III`, and `Taxon_IV` should be applied to the tree of Figure 10.4, from left to right. We then instantiate the tree using the new constructor. We now have *three* constructors for our `Tree` class:

- `Tree(void)`, the default constructor that is implemented in the header file and made private so that it cannot be called.
- `Tree(double lambda, double mu, double duration, RandomVariable* rv)`, a constructor that sets up the tree under the stochastic birth-death process of cladogenesis.
- `Tree(std::string newickStr)`, a constructor that will eventually set up the tree reflected in the Newick string that is passed to it. So far, this implementation is incomplete.

Currently, this new constructor does one thing: it calls a function called `parseNewickString`. The string that is passed into the constructor contains the information about the tree, but we need to break it up, isolating the individual components. We will do different things when we build up the tree in computer memory when we encounter a left parentheses, right parentheses, comma, *etc.* The `parseNewickString` reads the string character-by-character and isolates the various components, returning the isolated components as a vector of strings. Each element of this vector represents an individual component of the Newick string.

Parsing is a real pain in the neck. The `parseNewickString` first declares a `vector` of strings. It then enters a loop, incremented by a variable `i`, that reads the Newick string one character at a time. If the character is one of the special ones — `() , : ;` — it immediately adds the character to the vector of strings. Otherwise, the character is part of a taxon name or a branch length. In this case, the parser races ahead to the next special symbol using a new type of loop (`while`), incrementing a new counter (`j`), adding the entire string (taxon name or branch length) to the vector.

The parser introduces a few new ideas. First of all, you can see in the conditional `if` statement that we use two parallel lines. This is read ‘or.’ You can check for several conditions in a `if` statement:

- `==` : is the thing on the left equal to the thing on the right (equal to)
- `!=` : is the thing on the left not equal to the thing on the right (not equal to)
- `||` : the statement on the left or the statement on the right is true (logical OR)
- `&&` : the statements to the left and to the right are both true (logical AND)

The parser also prints out the vector of isolated elements. It does this from between a compiler directive:

```
#  if defined(DEBUG_NEWICK_PARSER)

#  endif
```

It’s possible that we will want to go back to check that we have correctly parsed a string. This compiler directive allows us to turn on or turn off that bit of code by defining or undefining `DEBUG_NEWICK_PARSER`. When I run the program, I get the following output:

The Newick string, broken into its parts:

```
tk[0] = "("
tk[1] = "Taxon_I"
tk[2] = ":"
tk[3] = "0.3"
tk[4] = ","
tk[5] = "("
tk[6] = "("
tk[7] = "Taxon_II"
tk[8] = ":"
tk[9] = "0.1"
tk[10] = ","
tk[11] = "Taxon_III"
tk[12] = ":"
tk[13] = "0.1"
tk[14] = ")"
tk[15] = ":"
tk[16] = "0.1"
```

```

    tks[17] = ","
    tks[18] = "Taxon_IV"
    tks[19] = ":"
    tks[20] = "0.2"
    tks[21] = ")"
    tks[22] = ":"
    tks[23] = "0.1"
    tks[24] = ")"
    tks[25] = ";"

```

Program ended with exit code: 0

Check that your output looks like this. If so, congratulations! You successfully parsed the Newick string. Now that you have confirmed that we have successfully parsed the string, turn off the loop that prints the vector of parsed elements by changing the statement at the top of the file to:

Example

```
#undef DEBUG_NEWICK_PARSER
```

Now, the segment of code is not even seen by the compiler.

The complete implementation of our new constructor is shown here:

Example

```

Tree::Tree(std::string newickStr) {

    // break the string into its component parts
    std::vector<std::string> tokens = parseNewickString(newickStr);

    // build up the tree from the parsed Newick string
    bool readingBranchLength = false;
    Node* p = NULL;
    for (int i=0; i<tokens.size(); i++)
    {
        std::string token = tokens[i];
        //std::cout << token << std::endl;
        if (token == "(")
        {
            // new node

```

```
Node* newNode = new Node;
nodes.push_back(newNode);
if (p == NULL)
    root = newNode;
else
{
    newNode->setAnc(p);
    if (p->getLft() == NULL)
        p->setLft(newNode);
    else
        p->setRht(newNode);
}
p = newNode;
}
else if (token == ")" || token == ",")
{
    // move down one node
    if (p->getAnc() == NULL)
    {
        std::cout << "Error: We cannot find an expected ancestor" << std::endl;
        exit(1);
    }
    p = p->getAnc();
}
else if (token == ":")
{
    // begin reading a branch length
    readingBranchLength = true;
}
else if (token == ";")
{
    // finished!
    if (p != root)
    {
        std::cout << "Error: We expect to finish at the root node" << std::endl;
        exit(1);
    }
}
else
{

```

```

        // we have a taxon name or a branch length
        if (readingBranchLength == true)
        {
            double x = stod(token);
            p->setBranchLength(x);
            readingBranchLength = false;
        }
        else
        {
            Node* newNode = new Node;
            nodes.push_back(newNode);
            newNode->setAnc(p);
            if (p->getLft() == NULL)
                p->setLft(newNode);
            else
                p->setRht(newNode);
            newNode->setName(token);
            p = newNode;
        }
    }
}

// index the nodes
int ndeIdx = 0;
for (int i=0; i<nodes.size(); i++)
{
    if (nodes[i]->getLft() == NULL)
        nodes[i]->setIndex(ndeIdx++);
}
for (int i=0; i<nodes.size(); i++)
{
    if (nodes[i]->getLft() != NULL)
        nodes[i]->setIndex(ndeIdx++);
}

// initialize the postorder traversal sequence
initializeTraversalOrder();
}

```

Walk through this code line-by-line. You will see that we have a loop over the parsed elements in

the Newick string. I call these parsed elements **token**. We do different things depending on the identity of the **token**:

- **token == "("** — The token is a left parentheses, in which case we add a new node. We also set up the pointers for this new node and the ancestor of this node, if there is one. The first time we create a node, there are no other nodes that can be neighbors (to the left, right, or ancestrally). This first node is the **root** node.
- **token == ")"** || **token == ","** — The token is either a right parentheses or a comma. In either case, we move to the ancestor of the current point in the tree.
- **token == ":"** — The token is a colon, which means the next token is a branch length. We set a **bool** variable named **readingBranchLength** to be true.
- **token == ";"** — The token is a semicolon, in which case we should be finished reading all of the Newick string elements. The algorithm works such that we should end at the root. We check that this is true. If it isn't, we exit the program abruptly. Something is very wrong with the Newick string that was passed to the constructor.
- If none of the above are true, then the token must be either a taxon name or a branch length. If it is a taxon name, we add a new node, just as we did when we encountered a left parentheses, but this time we also add the taxon name information. If the token is a branch length, we simply set the branch length for the node we are currently pointed at.

After we build up the tree, we index the nodes such that the tips are labeled from $0, 1, \dots, N - 1$ and the interior nodes are assigned the indices $N, N + 1, \dots$. Finally, we initialize the postorder traversal sequence for the tree.

Check that the tree accurately represents the Newick string that was passed in. You can modify the **main** function to read:

Example

```
int main(int argc, char* argv[]) {

    RandomVariable rv;

    std::string myTree = "(Taxon_I:0.3,((Taxon_II:0.1,Taxon_III:0.1):0.1,Taxon_IV:0.2):0.1,Taxon_V:0.2):0.3";
    Tree t(myTree);
    t.listNodes();
    std::cout << t.getNewickRepresentation() << std::endl;

    return 0;
}
```

If you run the program, you will note that the Newick string representation does not look right. This is because we begin traversing the tree from the node to the left of the root node. Modify the `getNewickRepresentation` function to read:

Example

```
std::string Tree::getNewickRepresentation(void) {  
  
    std::stringstream ss;  
    if (root->getLft() != NULL && root->getRht() != NULL)  
        writeTree(root, ss);  
    else  
        writeTree(root->getLft(), ss);  
    std::string newick = ss.str();  
    return newick;  
}
```

When I run the program, I get the following output:

```
Node 4 (0x100503780)  
  Lft:  0x100508e90  
  Rht:  0x100505c90  
  Anc:  0x0  
  Name:  ""  
  Brlen: 0  
  Time: 0  
Node 0 (0x100508e90)  
  Lft:  0x0  
  Rht:  0x0  
  Anc:  0x100503780  
  Name:  "Taxon_I"  
  Brlen: 0.3  
  Time: 0  
Node 5 (0x100505c90)  
  Lft:  0x1005043f0  
  Rht:  0x100503020  
  Anc:  0x100503780  
  Name:  ""  
  Brlen: 0.1  
  Time: 0  
Node 6 (0x1005043f0)  
  Lft:  0x100507300
```

```

    Rht: 0x100503580
    Anc: 0x100505c90
    Name: ""
    Brlen: 0.1
    Time: 0
Node 1 (0x100507300)
    Lft: 0x0
    Rht: 0x0
    Anc: 0x1005043f0
    Name: "Taxon_II"
    Brlen: 0.1
    Time: 0
Node 2 (0x100503580)
    Lft: 0x0
    Rht: 0x0
    Anc: 0x1005043f0
    Name: "Taxon_III"
    Brlen: 0.1
    Time: 0
Node 3 (0x100503020)
    Lft: 0x0
    Rht: 0x0
    Anc: 0x100505c90
    Name: "Taxon_IV"
    Brlen: 0.2
    Time: 0
(Taxon_I:0.30000,((Taxon_II:0.10000,Taxon_III:0.10000):0.10000,Taxon_IV:0.20000):0.10000)
Program ended with exit code: 0

```

All looks well to me. We have successfully modified our `Tree` class to make a tree from a Newick string. Now we need to actually simulate DNA sequences on this tree. Onward! Upward!

10.3.2 Creating the Alignment class

The alignment class will take as input a tree and parameters for the substitution model. It will create a character matrix, internally, of the sites at the tips of the tree.

We need to create the files for our new `Alignment` class. Using your IDE, create the new class, which should result in two files: `Alignment.hpp` and `Alignment.cpp`. We will modify the `main` function and begin implementing the `Alignment` class. First, add the following code to the `Alignment` header file (`Alignment.hpp`):

Example

```

#ifndef Alignment_hpp
#define Alignment_hpp

class RandomVariable;
class Tree;

class Alignment {

public:
    Alignment(Tree* t, RandomVariable* rv, int ns, double ep[6], double bfp[4]);

private:
    void    initializeRateMatrix(double ep[6], double bfp[4]);
    void    scaleRateMatrix(double bfp[4]);
    int     numSites;
    double  q[4][4];
};

#endif

```

Implement the three functions in the `Alignment.cpp` file. First, add the header information and the constructor:

Example

```

#include <iomanip>
#include <iostream>
#include "Alignment.hpp"

#define DEBUG_RATE_MATRIX

Alignment::Alignment(Tree* t, RandomVariable* rv, int ns, double ep[6], double bfp[4]) {

    // set the instance variable
    numSites = ns;
}

```

```

        // initialize and scale the rate matrix
        initializeRateMatrix(ep, bfp);
        scaleRateMatrix(bfp);

    }

```

The constructor sets a single instance variable (`numSites`, the length of the DNA sequences to be simulated) and initializes a matrix, `q` which is another instance variable of the class and will hold the substitution rates. The function `initializeRateMatrix` initializes the rate matrix. It takes as input the six exchangeability parameters and four base frequency (or stationary probability) parameters of the general time reversible model of DNA substitution (the GTR model; Tavaré, 1986). The other function, `scaleRateMatrix`, rescales the substitution rate matrix such that the average rate of substitution is one.

Make certain to implement the `initializeRateMatrix` and `scaleRateMatrix` functions, too. Here is the code for the `initializeRateMatrix` function:

Example

```

void Alignment::initializeRateMatrix(double ep[6], double bfp[4]) {

    // the exchangeability parameters in ep are in the
    // order A <-> C, A <-> G, A <-> T, C <-> G, C <-> T,
    // and G <-> T. The base frequency, or stationary
    // probability, parameters are in the order A, C, G, and T.

    // set the off-diagonal components
    for (int i=0, k=0; i<4; i++)
    {
        for (int j=i+1; j<4; j++)
        {
            q[i][j] = ep[k] * bfp[j];
            q[j][i] = ep[k] * bfp[i];
            k++;
        }
    }

    // set the diagonal components

```

```

for (int i=0; i<4; i++)
{
    double sum = 0.0;
    for (int j=0; j<4; j++)
    {
        if (i != j)
            sum += q[i][j];
    }
    q[i][i] = -sum;
}

# if defined(DEBUG_RATE_MATRIX)
std::cout << "Rate matrix before scaling:" << std::endl;
for (int i=0; i<4; i++)
{
    for (int j=0; j<4; j++)
    {
        if (q[i][j] > 0.0)
            std::cout << std::fixed << std::setprecision(3) << " " << q[i][j] << " ";
        else
            std::cout << std::fixed << std::setprecision(3) << q[i][j] << " ";
    }
    std::cout << std::endl;
}
# endif
}

```

Don't forget the code for rescaling the rate matrix:

Example

```

void Alignment::scaleRateMatrix(double bfp[4]) {

    // rescale the rate matrix such that the average
    // rate of substitution is one
    double averageRate = 0.0;
    for (int i=0; i<4; i++)

```

```

        averageRate += -(bfp[i] * q[i][i]);

double scaler = 1.0 / averageRate;

for (int i=0; i<4; i++)
    for (int j=0; j<4; j++)
        q[i][j] *= scaler;

# if defined(DEBUG_RATE_MATRIX)
std::cout << "Rate matrix after scaling:" << std::endl;
for (int i=0; i<4; i++)
{
    for (int j=0; j<4; j++)
    {
        if (q[i][j] > 0.0)
            std::cout << std::fixed << std::setprecision(3) << " " << q[i][j] << " ";
        else
            std::cout << std::fixed << std::setprecision(3) << q[i][j] << " ";
    }
    std::cout << std::endl;
}
# endif
}

```

Both the `initializeRateMatrix` and `scaleRateMatrix` functions have nicely-formatted debugging statements behind a compiler directive. The debugging can be turned on or off by defining (`define`) or undefining (`undef`) the `DEBUG_RATE_MATRIX` constant.

We want to test what we have written so far. Go to the file `main.cpp` and modify the code in that file to read:

Example

```

#include <iostream>
#include <string>
#include "Alignment.hpp"
#include "RandomVariable.hpp"
#include "Tree.hpp"

```

```

int main(int argc, char* argv[]) {

    // instantiate the random number generator
    RandomVariable rv;

    // set the parameters of the simulation
    int numSites = 100;
    std::string myTree = "(Taxon_I:0.3,((Taxon_II:0.1,Taxon_III:0.1):
                          0.1,Taxon_IV:0.2):0.1);";
    double exchangeabilityParms[6] = { 1.0, 5.0, 1.0, 1.0, 5.0, 1.0 };
    double baseFrequencyParms[4] = { 0.4, 0.3, 0.2, 0.1 };

    // instantiate the tree and alignment objects
    Tree t(myTree);
    Alignment myAlignment(&t, &rv, numSites, exchangeabilityParms, baseFrequencyParms);

    return 0;
}

```

Compile and run the code. I get the output:

```

Rate matrix before scaling:
-1.400  0.300  1.000  0.100
 0.400 -1.100  0.200  0.500
 2.000  0.300 -2.400  0.100
 0.400  1.500  0.200 -2.100
Rate matrix after scaling:
-0.886  0.190  0.633  0.063
 0.253 -0.696  0.127  0.316
 1.266  0.190 -1.519  0.063
 0.253  0.949  0.127 -1.329
Program ended with exit code: 0

```

Note that the last 4×4 matrix that is printed is identical to the rate matrix that we used to illustrate DNA simulation earlier in the chapter. That rate matrix was specified under the HKY85 model

$$\mathbf{Q} = \{q_{ij}\} = \begin{pmatrix} - & \pi_C & \kappa\pi_G & \pi_T \\ \pi_A & - & \pi_G & \kappa\pi_T \\ \kappa\pi_A & \pi_C & - & \pi_T \\ \pi_A & \kappa\pi_C & \pi_G & - \end{pmatrix} \mu$$

with parameter values: $\kappa = 5$, $\pi_A = 0.4$, $\pi_C = 0.3$, $\pi_G = 0.2$, and $\pi_T = 0.1$. Our program will simulate DNA sequences under the more general GTR model, that has rate matrix

$$\mathbf{Q} = \{q_{ij}\} = \begin{pmatrix} - & r_{AC}\pi_C & r_{AG}\pi_G & r_{AT}\pi_T \\ r_{AC}\pi_A & - & r_{CG}\pi_G & r_{CT}\pi_T \\ r_{AG}\pi_A & r_{CG}\pi_C & - & r_{GT}\pi_T \\ r_{AT}\pi_A & r_{CT}\pi_C & r_{GT}\pi_G & - \end{pmatrix} \mu$$

The parameters of the GTR model are divided into the exchangeability parameters and base frequency parameters, $\mathbf{r} = (r_{AC}, r_{AG}, r_{AT}, r_{CG}, r_{CT}, r_{GT})$ and $\pi = (\pi_A, \pi_C, \pi_G, \pi_T)$, respectively. In the `main` function we initialize the exchangeability and base frequency parameters to be equal to:

$$\begin{aligned} \mathbf{r} &= (1, 5, 1, 1, 5, 1) \\ \pi &= (0.4, 0.3, 0.2, 0.1) \end{aligned}$$

Can you see why our choice for the GTR parameters is equivalent to the parameterization of the HKY85 model we used earlier?

We will simulate the DNA sequences one site at a time. We begin at the root and visit the other nodes in preorder (*i.e.*, from the root to the tips). When we visit a node, we can be assured that we know the nucleotide state of its ancestor. We will then simulate the DNA sequences one branch at a time.

We will make another incremental change to the new constructor. Modify your code so the constructor reads:

Example

```
Alignment::Alignment(Tree* t, RandomVariable* rv, int ns, double ep[6], double bfp[4]) {

    // set the instance variable
    numSites = ns;

    // initialize and scale the rate matrix
    initializeRateMatrix(ep, bfp);
    scaleRateMatrix(bfp);

    // get the traversal order of the nodes for the tree
    std::vector<Node*> traversalSequence = t->getTraversalOrder();

    // allocate enough memory to hold the DNA sequences for each node in the tree
    int** nodeSequences = new int*[traversalSequence.size()];
    nodeSequences[0] = new int[traversalSequence.size() * numSites];
    for (int i=1; i<traversalSequence.size(); i++)
        nodeSequences[i] = nodeSequences[i-1] + numSites;
```

```

    for (int i=0; i<traversalSequence.size(); i++)
        for (int j=0; j<numSites; j++)
            nodeSequences[i][j] = 0;

    // loop over the sites
    for (int c=0; c<numSites; c++)
    {
        // traverse the nodes in preorder, simulating along each branch
        for (int n=(int)traversalSequence.size()-1; n>=0; n--)
        {

        }
    }

    // allocate a matrix for just the tip nodes and fill it in

    // free the memory that was allocated
    delete [] nodeSequences[0];
    delete [] nodeSequences;
}

```

As before, we initialize the `numSites` instance variable and set up the rate matrix, `q`. Now, however, we do several new things. First, we get from the tree its postorder traversal sequence. The vector we obtained a reference to not only contains the memory addresses of the nodes in the correct order (so we can traverse the tree in post- or preorder) but we now know how many nodes there are in the tree. The next portion of the code is likely to be confusing.

We dynamically allocate memory that can be accessed as a *matrix*, with two sets of square brackets. We want to be able to quickly find the nucleotide assigned to any node in the tree (after the simulation is complete). For example, if we wanted to access the nucleotide for node 5 and site 4904, we should be able to do something like,

```
int nuc = nodeSequences[5][4904];
```

The problem is that we cannot dynamically allocate the memory for a matrix. Rather, all we can allocate is an array (vector). Keep in mind that the `new` function returns the memory address of the first element of the array. In this segment of code, we first allocate a vector of `int` pointers (`int*`). You can see that we allocate an array of `int*` by looking at the segment of code `new int* [<number of nodes>]`. The `new` function must return a pointer to the first element. Because the elements we just allocated are all `int*`, we must declare a variable of type `int**` to hold the memory address. The next thing we do is allocate enough `int` variables to hold the entire matrix. The size will be the number of nodes times the sequence length. Because we are allocating an

array of `int` variables, the `new` function will return a pointer to the first element; the return type is `int*`. We then initialize the remaining values setting up the matrix. Figure 10.6 illustrates this the general idea, showing visually how the two calls to the `new` function can act together to allocate memory that can be accessed as if it were a matrix.

The two calls to `new` in which memory was allocated are matched, in reverse order, at the end of the constructor where the memory is freed.

The guts of the simulation will occur in the nested loops. One of the loops is over sites and the other over the nodes of the tree. Note that we will be visiting the elements of the postorder traversal sequence in reverse order. This is equivalent to visiting the nodes in preorder (from the root to the tips of the tree).

Now that we have a way to remember the nucleotides (A, C, G, or T) that are assigned to the nodes of the tree, let's complete the simulation. Enter the following code segment in the appropriate place in the constructor:

Example

```
// loop over the sites
```

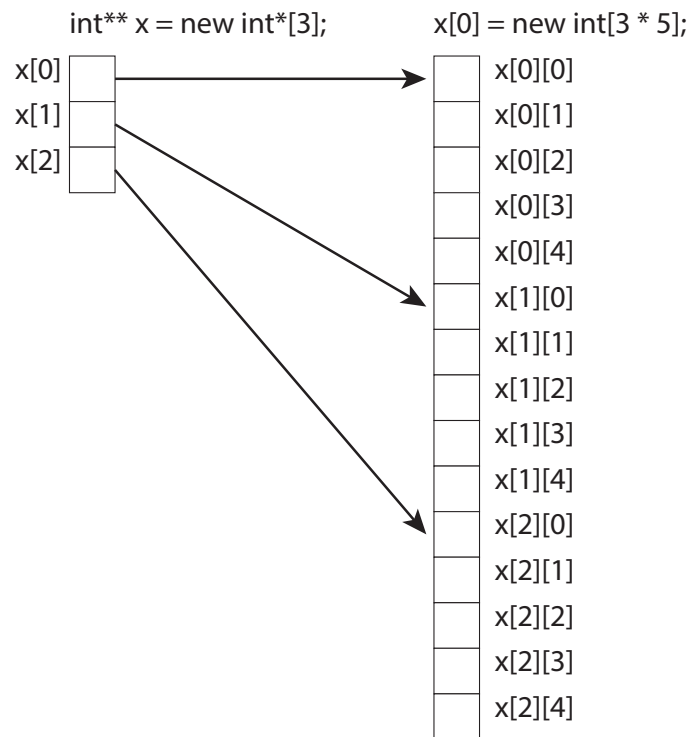


Figure 10.6: An example of how a 3×5 matrix can be dynamically allocated. The `new` function must be called twice.


```

for (int c=0; c<numSites; c++)
{
    // traverse the nodes in preorder, simulating along each branch
    for (int n=(int)traversalSequence.size()-1; n>=0; n--)
    {
        Node* p = traversalSequence[n];
        if (p->getAnc() == NULL)
        {
            // we are at the root of the tree
            double u = rv->uniformRv(), sum = 0.0;
            for (int i=0; i<4; i++)
            {
                sum += bfp[i];
                if (u < sum)
                {
                    nodeSequences[p->getIndex()][c] = i;
                    break;
                }
            }
        }
        else
        {
            // we are at an internal node or a tip
            int curNuc = nodeSequences[p->getAnc()->getIndex()][c];
            double duration = p->getBranchLength();
            double t = 0.0;
            while (t < duration)
            {
                double rate = -q[curNuc][curNuc];
                t += rv->exponentialRv(rate);
                if (t < duration)
                {
                    // we have a change...what kind is it?
                    double u = rv->uniformRv(), sum = 0.0;
                    for (int j=0; j<4; j++)
                    {
                        if (j != curNuc)
                        {
                            sum += q[curNuc][j] / rate;
                            if (u < sum)

```

```

        {
            curNuc = j;
            break;
        }
    }
}
nodeSequences[p->getIndex()][c] = curNuc;
}
}
}

```

You should include the `Tree`, `Node`, and `RandomVariable` header files in this file because we make use of functionality from all of those classes. The method we use to simulate is very similar to, if not identical to, the heart of the simulation engine we used for the birth-death process. We start the simulation at one end of the branch. We continue until the simulation time has exceeded the branch length. We use the rate matrix to parameterize the exponential distribution that determines how long the process waits between substitutions and also to calculate the probabilities of the various types of changes when the process experiences a change.

At this point, we have completed the simulation. Unfortunately, we have no way to visualize the results. Let's do that next by completing the implementation of the `Alignment` constructor. First, add to the header file (`Alignment.hpp`) three instance variables

Example

```

int            numTaxa;
int**          matrix;
std::vector<std::string> names;

```

and a destructor and a `print` function:

Example

```

        ~Alignment(void);
void      print(void);

```

After doing this, your header file for the `Alignment` class should look like:

Example

```

#ifndef Alignment_hpp
#define Alignment_hpp

#include <vector>
#include <string>
class RandomVariable;
class Tree;

class Alignment {

    public:

        Alignment(Tree* t, RandomVariable* rv, int ns, double ep[6],
        ~Alignment(void);
        void      print(void);

    private:
        void      initializeRateMatrix(double ep[6], double bfp[4]);
        void      scaleRateMatrix(double bfp[4]);
        int       numTaxa;
        int       numSites;
        double    q[4][4];
        int**     matrix;
        std::vector<std::string> names;
};

#endif

```

Now that you have completed the modification to the header file, enter the following code to the constructor at the correct position in the code:

Example

```

// allocate a matrix for just the tip nodes and fill it in
numTaxa = 0;
for (int i=0; i<traversalSequence.size(); i++)
{
    if (traversalSequence[i]->getLft() == NULL)
        numTaxa++;
}

matrix = new int*[numTaxa];
matrix[0] = new int[numTaxa * numSites];
for (int i=1; i<numTaxa; i++)
    matrix[i] = matrix[i-1] + numSites;
for (int i=0; i<numTaxa; i++)
    for (int j=0; j<numSites; j++)
        matrix[i][j] = 0;

for (int n=0; n<traversalSequence.size(); n++)
{
    Node* p = traversalSequence[n];
    if (p->getLft() == NULL)
    {
        names.push_back(p->getName());
        int idx = p->getIndex();
        for (int c=0; c<numSites; c++)
            matrix[idx][c] = nodeSequences[idx][c];
    }
}

```

Don't forget to implement the destructor, that simply deletes the `matrix` instance variable:

Example

```

Alignment::~Alignment(void) {

    delete [] matrix[0];
    delete [] matrix;
}

```

```
}
```

The last part is to implement the `print` function. Do that now by adding the following code to your implementation file (`Alignment.cpp`):

Example

```
void Alignment::print(void) {

    // get length of longest name
    int length = 0;
    for (int i=0; i<names.size(); i++)
    {
        if (names[i].size() > length)
            length = (int)names[i].size();
    }

    // print the alignment
    for (int i=0; i<numTaxa; i++)
    {
        std::cout << "    " << names[i];
        for (int j=0; j<length-names[i].size(); j++)
            std::cout << " ";
        std::cout << " ";
        for (int j=0; j<numSites; j++)
            std::cout << matrix[i][j];
        std::cout << std::endl;
    }
}
```

Before you compile and run the program, add a statement that will print the matrix in the `main` function,

Example

```
myAlignment.print();
```

When I ran the program, I saw the following output:

```
Rate matrix before scaling:
-1.400  0.300  1.000  0.100
 0.400 -1.100  0.200  0.500
 2.000  0.300 -2.400  0.100
 0.400  1.500  0.200 -2.100
Rate matrix after scaling:
-0.886  0.190  0.633  0.063
 0.253 -0.696  0.127  0.316
 1.266  0.190 -1.519  0.063
 0.253  0.949  0.127 -1.329
Taxon_I    10100213021022001001332000010...310030223101120210001202020111020222202313202
Taxon_II   20101013201020201103030000010...31203302312112013022020102010100022222333001
Taxon_III  32101013221022001101032000010...310031023121120130021203020131000202202313001
Taxon_IV   30101010201020021101002200010...110033220121100130221002220131000222220113201
Program ended with exit code: 0
```

Hmmm. No As, Cs, Gs, or Ts. Only 0s, 1s, 2s, and 3s. Let's make one more change to the code. First, add a function to the `Alignment` implementation file:

Example

```
char Alignment::convertIndexToNucleotide(int x) {

    char nucs[4] = { 'A', 'C', 'G', 'T' };
    return nucs[x];
}
```

Don't forget to add the function profile in the header file. I would declare it as `protected`. Now that this little function has been implemented, modify one line of the `print` function to read:

Example

```
std::cout << convertIndexToNucleotide( matrix[i][j] );
```

While you are at it, you might as well turn off the debugging statements for the rate matrix. When I run the program with this slight modification, I get:

```
Taxon_I    TTTAAAACACGCGGCGAAAGACGAGACCA...CCGCAACCAGAATCCGCCCTACAACCCGGAGGGAAAACCGAAATCGGCGGC
Taxon_II   TATCAAGTACACCACCAACGAGGGGACCA...CCGCGACCAAGATCCGCCCTCCAGCCCCAAAAAAAAAACCGGAATCAACCGT
Taxon_III  TATCGAGTGCTCCACCAGTGAGGGGACCA...ACGCAACC AAAATCCGCCCTACAGCCCAAGGCAAAGACCGGAATCAACCGT
Taxon_IV   TCTCAAGCAAACCGCCAAAGAAGGAATAA...CCACAGCCGAGATCCGCCCCGGAGCCCCGAAAGAAAACCGGAATAAACGGT
Program ended with exit code: 0
```

This looks more like it: An alignment of nucleotide sequences.

10.3.3 Using PAUP* to analyze the data

10.3.4 How well do alternative methods work?

Contents

Appendices

Appendix A

Random Variable Class

main.cpp file from Chapter 6.6

```
#include <iomanip>
#include <iostream>
#include "RandomVariable.hpp"

int main(int argc, const char * argv[]) {

    // instantiate the RandomVariable class
    RandomVariable rv;

    /* Generate uniform(a,b) random variables. Note
       that the mean of a uniform(a,b) distribution is
        $0.5 * (a + b)$  and the variance is  $(1/12) * (b-a) * (b-a)$  */
    int numUniforms = 10000;
    double a = 0.0, aOld = 0.0, s = 0.0;
    for (int i=0; i<numUniforms; i++)
    {
        double u = rv.uniformRv();
        if (i == 0)
        {
            a = u;
            s = 0.0;
        }
        else
        {
            aOld = a;
            a = aOld + (u - aOld) / (i+1);
        }
    }
}
```

```

        s = s + (u - aOld) * (u - aOld);
    }
}
double mean = a;
double variance = 0.0;
if (numUniforms > 1)
    variance = s / (numUniforms - 1);

std::cout << std::fixed << std::setprecision(3);
std::cout << "Results for " << numUniforms << " uniform(0,1) random variables" << std::endl;
std::cout << "    Mean:      " << mean << " (Expected=" << 0.5 << ")" << std::endl;
std::cout << "    Variance: " << variance << " (Expected=" << (1.0/12.0) << ")" << std::endl;

/* Generate exponential(lambda) random variables. Note
   that the mean of an exponential(lambda) distribution is
   1.0/lambda and the variance is 1.0/(lambda*lambda). */
int numExponentials = 10000;
double lambda = 10.0;
a = 0.0;
aOld = 0.0;
s = 0.0;
for (int i=0; i<numExponentials; i++)
{
    double u = rv.exponentialRv(lambda);
    if (i == 0)
    {
        a = u;
        s = 0.0;
    }
    else
    {
        aOld = a;
        a = aOld + (u - aOld) / (i+1);
        s = s + (u - aOld) * (u - aOld);
    }
}
mean = a;
variance = 0.0;
if (numUniforms > 1)
    variance = s / (numExponentials - 1);

std::cout << "Results for " << numExponentials << " exponential(" << lambda << ") random variables" << std::endl;
std::cout << "    Mean:      " << mean << " (Expected=" << 1.0/lambda << ")" << std::endl;
std::cout << "    Variance: " << variance << " (Expected=" << 1.0/(lambda*lambda) << ")" << std::endl;

```

```

    return 0;
}

```

RandomVariable.hpp file from Chapter 6.6

```

#ifndef RandomVariable_hpp
#define RandomVariable_hpp

class RandomVariable {

    public:
        RandomVariable(void);
        RandomVariable(int x);
        double exponentialRv(double lambda);
        double uniformRv(void);
        double uniformRv(double lower, double upper);

    protected:
        int seed;
};

#endif

```

RandomVariable.cpp file from Chapter 6.6

```

#include <cmath>
#include <ctime>
#include <iostream>
#include "RandomVariable.hpp"

RandomVariable::RandomVariable(void) {

    seed = (int)time(NULL);
}

RandomVariable::RandomVariable(int x) {

```

```
    seed = x;
}

double RandomVariable::exponentialRv(double lambda) {

    return -log(uniformRv()) / lambda;
}

double RandomVariable::uniformRv(void) {

    int hi = seed / 127773;
    int lo = seed % 127773;
    int test = 16807 * lo - 2836 * hi;
    if (test > 0)
        seed = test;
    else
        seed = test + 2147483647;
    return (double)(seed) / (double)2147483647;
}

double RandomVariable::uniformRv(double lower, double upper) {

    return 0.0;
}
```

Appendix B

MCMC Coin Tossing Program

main.cpp file from Chapter 7.4

```
#include <cmath>
#include <iomanip>
#include <iostream>
#include "RandomVariable.hpp"

int main(int argc, char* argv[]) {

    // instantiate an object for generating random numbers
    RandomVariable rv;

    // this user-interface sucks!
    int numTosses      = 100;
    int numHeads       = 43;
    int numTails       = numTosses - numHeads;
    int chainLength    = 1000000;
    int printFrequency = 1000;
    int sampleFrequency = 1;
    double window      = 0.1;

    // initialize theta to some value
    double theta = rv.uniformRv();

    // run the Markov chain
    int bins[100];
    for (int i=0; i<100; i++)
        bins[i] = 0;

    // here we set up a vector of ints to
    // keep track of how often each between
    // interval 0 and 1 was visited
```



```

int numSamples = 0;
for (int n=1; n<=chainLength; n++)
{
    // propose a new value for theta using a sliding window
    // proposal mechanism
    double thetaPrime = theta + (rv.uniformRv() - 0.5) * window;
    if (thetaPrime < 0.0)
        thetaPrime = -thetaPrime;
    else if (thetaPrime > 1.0)
        thetaPrime = 2.0 - thetaPrime;

    // calculate the probability of accepting thetaPrime as
    // the next state of the chain
    double lnLikelihoodRatio = (numHeads*log(thetaPrime) + numTails*log(1.0-thetaPrime)) -
                               (numHeads*log(theta) + numTails*log(1.0-theta));
    double lnPriorRatio = 0.0; // natural log of 1.0
    double lnHastingsRatio = 0.0; // natural log of 1.0
    double lnR = lnLikelihoodRatio + lnPriorRatio + lnHastingsRatio;
    double R = 0.0;
    if (lnR < -300.0) // we go through this rigamarole to avoid underflow
        R = 0.0; // when we exponentiate lnR
    else if (lnR > 0.0)
        R = 1.0;
    else
        R = exp(lnR);

    // print (part 1)
    if (n % printFrequency == 0)
    {
        std::cout << std::setw(5) << n << " -- ";
        std::cout << std::fixed << std::setprecision(3) << theta << " -> ";
        std::cout << std::fixed << std::setprecision(3) << thetaPrime << " ";
    }

    // accept or reject thetaPrime, and update the state of the chain
    double u = rv.uniformRv();
    bool isAccepted = false;
    if (u < R)
    {
        theta = thetaPrime;
        isAccepted = true;
    }

    // print (part 2)
    if (n % printFrequency == 0)

```

```

    {
        if (isAccepted == true)
            std::cout << "(Accepted)";
        else
            std::cout << "(Rejected)";
        std::cout << std::endl;
    }

    // sample the chain
    if (n % sampleFrequency == 0)
    {
        numSamples++; // note that when we cast theta*100.0,
        bins[(int)(theta*100.0)]++; // which is a number between 0.000...001
        // and 99.9999...999, we lose the decimal
        // portion (i.e., 99.99999 goes to 99)
    }

    // summarize the results
    double cumulativeProbability = 0.0;
    for (int i=0; i<100; i++)
    {
        double intervalProbability = (double)bins[i] / numSamples;
        cumulativeProbability += intervalProbability;
        std::cout << std::fixed << std::setprecision(2);
        std::cout << i * 0.01 << " - " << (i+1) * 0.01 << " -- ";
        std::cout << std::setw(5) << bins[i] << " ";
        std::cout << std::fixed << std::setprecision(3);
        std::cout << intervalProbability << " ";
        std::cout << cumulativeProbability << " ";
        std::cout << std::endl;
    }

    return 0;
}

```


Appendix C

A Simple Tree

main.cpp file from Chapter 8.4

```
#include <iostream>
#include "Node.hpp"
#include "Tree.hpp"

int main(int argc, char* argv[]) {

    // instantiate the tree
    Tree t;

    // list the nodes
    t.listNodes();

    // get the postorder node sequence
    std::vector<Node*> postOrd = t.getTraversalOrder();

    // print the nodes in postorder
    std::cout << "Postorder: ";
    for (int i=0; i<postOrd.size(); i++)
        std::cout << postOrd[i]->getIndex() << " ";
    std::cout << std::endl;

    // print the nodes in preorder
    std::cout << "Preorder: ";
    for (int i=postOrd.size()-1; i>=0; i--)
        std::cout << postOrd[i]->getIndex() << " ";
    std::cout << std::endl;
```

```
    return 0;
}
```

Node.hpp file from Chapter 8.4

```
#ifndef Node_hpp
#define Node_hpp

#include <string>

class Node {

    public:

        Node(void);
        Node*    getLft(void) { return left; }
        Node*    getRht(void) { return right; }
        Node*    getAnc(void) { return ancestor; }
        int      getIndex(void) { return index; }
        std::string getName(void) { return name; }
        double   getBranchLength(void) { return branchLength; }
        void     setLft(Node* p) { left = p; }
        void     setRht(Node* p) { right = p; }
        void     setAnc(Node* p) { ancestor = p; }
        void     setIndex(int x) { index = x; }
        void     setName(std::string s) { name = s; }
        void     setBranchLength(double x) { branchLength = x; }
        void     print(void);

    protected:
        Node*    left;
        Node*    right;
        Node*    ancestor;
        int      index;
        std::string name;
        double   branchLength;
};

#endif
```

Node.cpp file from Chapter 8.4

```
#include <iostream>
#include "Node.hpp"

Node::Node(void) {

    left      = NULL;
    right     = NULL;
    ancestor  = NULL;
    index     = 0;
    name      = "";
    branchLength = 0.0;
}

void Node::print(void) {

    std::cout << "Node " << index << " (" << this << ")" << std::endl;
    std::cout << "  Lft:  " << left << std::endl;
    std::cout << "  Rht:  " << right << std::endl;
    std::cout << "  Anc:  " << ancestor << std::endl;
    std::cout << "  Name: \"\" << name << \"\" << std::endl;
    std::cout << "  Brlen: \"\" << branchLength << std::endl;
}
```

Tree.hpp file from Chapter 8.4

```
#ifndef Tree_hpp
#define Tree_hpp

#include <vector>
class Node;

class Tree {

public:

    Tree(void);
```

```

~Tree(void);
std::vector<Node*>& getTraversalOrder(void) { return postOrderSequence; }
void listNodes(void);

protected:
    void initializeTraversalOrder(void);
    void passDown(Node* p);
    Node* root;
    std::vector<Node*> nodes;
    std::vector<Node*> postOrderSequence;
};

#endif

```

Tree.cpp file from Chapter 8.4

```

#include <iostream>
#include "Node.hpp"
#include "Tree.hpp"

Tree::Tree(void) {

    // make the three-species tree of Figure 8.1

    // allocate the five nodes for the three-species tree
    for (int i=0; i<5; i++)
    {
        Node* newNode = new Node;
        nodes.push_back( newNode );
    }

    // set the member pointers
    Node* p = nodes[0];
    p->setAnc(nodes[3]);
    p->setLft(NULL);
    p->setRht(NULL);
    p->setIndex(0);
    p->setBranchLength(0.10);
    p->setName("Sp1");
}

```

```

    p = nodes[1];
    p->setAnc(nodes[3]);
    p->setLft(NULL);
    p->setRht(NULL);
    p->setIndex(1);
    p->setBranchLength(0.10);
    p->setName("Sp2");

    p = nodes[2];
    p->setAnc(nodes[4]);
    p->setLft(NULL);
    p->setRht(NULL);
    p->setIndex(2);
    p->setBranchLength(0.30);
    p->setName("Sp3");

    p = nodes[3];
    p->setAnc(nodes[4]);
    p->setLft(nodes[0]);
    p->setRht(nodes[1]);
    p->setIndex(3);
    p->setBranchLength(0.20);
    p->setName("");

    p = nodes[4];
    p->setAnc(NULL);
    p->setLft(nodes[3]);
    p->setRht(nodes[2]);
    p->setIndex(4);
    p->setBranchLength(0.0);
    p->setName("");

    root = nodes[4];
    initializeTraversalOrder();
}

Tree::~Tree(void) {

    for (int i=0; i<nodes.size(); i++)
        delete nodes[i];
}

void Tree::initializeTraversalOrder(void) {

    postOrderSequence.clear();

```



```
    passDown(root);
}

void Tree::listNodes(void) {

    for (int i=0; i<nodes.size(); i++)
    {
        nodes[i]->print();
    }
}

void Tree::passDown(Node* p) {

    if (p != NULL)
    {
        passDown(p->getLft());
        passDown(p->getRht());
        postOrderSequence.push_back(p);
    }
}
```

Appendix D

Birth-Death Process of Cladogenesis Code

main.cpp file from Chapter 9.3

```
#include <iostream>
#include "RandomVariable.hpp"
#include "Tree.hpp"

int main(int argc, char* argv[]) {

    RandomVariable rv;

    Tree t(3.0, 1.0, 1.0, &rv);
    std::cout << t.getNewickRepresentation() << std::endl;

    return 0;
}
```

Node.hpp file from Chapter 9.3

```
#ifndef Node_hpp
#define Node_hpp

#include <string>
```

```

class Node {

    public:

        Node(void);
        Node*    getLft(void) { return left; }
        Node*    getRht(void) { return right; }
        Node*    getAnc(void) { return ancestor; }
        int      getIndex(void) { return index; }
        std::string getName(void) { return name; }
        double    getBranchLength(void) { return branchLength; }
        double    getTime(void) { return time; }
        void      setLft(Node* p) { left = p; }
        void      setRht(Node* p) { right = p; }
        void      setAnc(Node* p) { ancestor = p; }
        void      setIndex(int x) { index = x; }
        void      setName(std::string s) { name = s; }
        void      setBranchLength(double x) { branchLength = x; }
        void      setTime(double x) { time = x; }
        void      print(void);

    protected:

        Node*    left;
        Node*    right;
        Node*    ancestor;
        int      index;
        std::string name;
        double    branchLength;
        double    time;
};

#endif

```

Node.cpp file from Chapter 9.3

```

#include <iostream>
#include "Node.hpp"

Node::Node(void) {

```

```

    left      = NULL;
    right     = NULL;
    ancestor  = NULL;
    index     = 0;
    name      = "";
    branchLength = 0.0;
    time      = 0.0;
}

void Node::print(void) {

    std::cout << "Node " << index << " (" << this << ")" << std::endl;
    std::cout << "  Lft:  " << left << std::endl;
    std::cout << "  Rht:  " << right << std::endl;
    std::cout << "  Anc:  " << ancestor << std::endl;
    std::cout << "  Name: \"" << name << "\"" << std::endl;
    std::cout << "  Brlen: " << branchLength << std::endl;
    std::cout << "  Time:  " << time << std::endl;
}

```

Tree.hpp file from Chapter 9.3

```

#ifndef Tree_hpp
#define Tree_hpp

#include <set>
#include <sstream>
#include <string>
#include <vector>
class Node;
class RandomVariable;

class Tree {

public:
    Tree(double lambda, double mu, double duration,
          RandomVariable* rv);
    ~Tree(void);
    std::vector<Node*>& getTraversalOrder(void) { return postOrderSequence; }
    void listNodes(void);
}

```

```

        std::string      getNewickRepresentation(void);
        int             getNumExtant(void) { return numExtant; }
        void             setNumExtant(int x) { numExtant = x; }

protected:
        Node*            Tree(void) { }
        void             chooseNodeFromSet(std::set<Node*>& s, RandomVariable* rv);
        void             initializeTraversalOrder(void);
        void             passDown(Node* p);
        void             writeTree(Node* p, std::stringstream& ss);
        Node*            root;
        std::vector<Node*> nodes;
        std::vector<Node*> postOrderSequence;
        int              numExtant;
};

#endif

```

Tree.cpp file from Chapter 9.3

```

#include <iomanip>
#include <iostream>
#include <set>
#include "Node.hpp"
#include "RandomVariable.hpp"
#include "Tree.hpp"

```

```

Tree::Tree(double lambda, double mu, double duration, RandomVariable* rv) {

    // generate a birth-death with parameter lambda, mu, and duration

    // initialize the single lineage, adding
    // the descendant to a list of active nodes
    nodes.push_back( new Node );
    nodes.push_back( new Node );
    nodes[0]->setLft(nodes[1]);
    nodes[1]->setAnc(nodes[0]);
    std::set<Node*> activeNodes;
    activeNodes.insert( nodes[1] );
    root = nodes[0];

```

```

// generate the full tree
double t = 0.0;
while (t < duration)
{
    // increment t using the exponential distribution
    double rate = activeNodes.size() * (lambda + mu);
    t += rv->exponentialRv(rate);

    // if t is still less than duration, go ahead do the speciation
    // or extinction thing on a randomly selected active lineage
    if (t < duration)
    {
        // choose a node
        Node* p = chooseNodeFromSet(activeNodes, rv);
        p->setTime(t);

        // choose a type of event
        double u = rv->uniformRv();
        if ( u < lambda / (lambda+mu) )
        {
            // speciation event
            Node* newLft = new Node;           // 1. allocate new left and
            Node* newRht = new Node;           //    right nodes
            nodes.push_back(newLft);           // 2. add the new nodes to the tree
            nodes.push_back(newRht);
            newLft->setAnc(p);                   // 3. set the ancestor of both new
            newRht->setAnc(p);                   //    nodes to be p
            p->setLft(newLft);                   // 4. set the left and right values of
            p->setRht(newRht);                   //    p to be the new nodes
            activeNodes.erase(p);               // 5. modify the list of active nodes
            activeNodes.insert(newLft);
            activeNodes.insert(newRht);
        }
        else
        {
            // extinction event
            activeNodes.erase(p);               // poor p ... he's dead
        }
    }
}

// clean up
numExtant = (int)activeNodes.size();
for (Node* nde : activeNodes)

```

```

    {
        nde->setTime(duration);
    }
initializeTraversalOrder();

// set the index variable and assign branch lengths from the node times
int nodeIdX = 0;
for (int i=0; i<postOrderSequence.size(); i++)
{
    Node* p = postOrderSequence[i];
    if (p->getLft() == NULL && p->getRht() == NULL)
    {
        p->setIndex(nodeIdX++);
        p->setName( std::to_string(nodeIdX) );
    }
    if (p->getAnc() != NULL)
        p->setBranchLength( p->getTime() - p->getAnc()->getTime() );
}
for (int i=0; i<postOrderSequence.size(); i++)
{
    Node* p = postOrderSequence[i];
    if ( !(p->getLft() == NULL && p->getRht() == NULL) )
        p->setIndex(nodeIdX++);
}
}

Tree::~Tree(void) {

    for (int i=0; i<nodes.size(); i++)
        delete nodes[i];
}

Node* Tree::chooseNodeFromSet(std::set<Node*>& s, RandomVariable* rv) {

    int whichNode = (int)(s.size() * rv->uniformRv());
    int i = 0;
    for (Node* nde : s)
    {
        if (whichNode == i)
            return nde;
        i++;
    }
    return NULL;
}

```

```

std::string Tree::getNewickRepresentation(void) {

    std::stringstream ss;
    if (root->getLft() != NULL && root->getRht() != NULL)
        writeTree(root, ss);
    else
        writeTree(root->getLft(), ss);
    std::string newick = ss.str();
    return newick;
}

void Tree::initializeTraversalOrder(void) {

    postOrderSequence.clear();
    passDown(root);
}

void Tree::listNodes(void) {

    for (int i=0; i<nodes.size(); i++)
    {
        nodes[i]->print();
    }
}

void Tree::passDown(Node* p) {

    if (p != NULL)
    {
        passDown(p->getLft());
        passDown(p->getRht());
        postOrderSequence.push_back(p);
    }
}

void Tree::writeTree(Node* p, std::stringstream& ss) {

    if (p != NULL)
    {
        if (p->getLft() == NULL)
        {
            ss << p->getName() << ":" << std::fixed << std::setprecision(5) << p->getBranch
        }
        else
        {

```



```
ss << "(";  
writeTree (p->getLft(), ss);  
ss << ",";  
writeTree (p->getRht(), ss);  
if (p->getAnc() == NULL)  
    ss << ")";  
else  
    ss << "):" << std::fixed << std::setprecision(5) << p->getBranchLength();  
}  
}
```

Appendix E

Site Probabilities

The probabilities of all 256 site patterns when simulated on the tree of Figure 10.4 under the HKY85 model of nucleotide substitution. The nucleotides for the site patterns are ordered from left to right (on the tree from Figure 10.4). These probabilities were calculated on the tree of Figure 10.4 under the HKY85 model of substitution with $\kappa = 5$, $\pi_A = 0.4$, $\pi_C = 0.3$, $\pi_G = 0.2$, and $\pi_T = 0.1$.

Patterns starting with A:

Pattern	Prob.	Pattern	Prob.	Pattern	Prob.	Pattern	Prob.
AAAA	0.199465	AGAA	0.014711	ACAA	0.004185	ATAA	0.001395
AAAC	0.004185	AGAC	0.000725	ACAC	0.005482	ATAC	0.000350
AAAG	0.014711	AGAG	0.019868	ACAG	0.000725	ATAG	0.000242
AAAT	0.001395	AGAT	0.000242	ACAT	0.000350	ATAT	0.001594
AACA	0.009075	AGCA	0.000843	ACCA	0.000703	ATCA	0.000121
AACC	0.000703	AGCC	0.000315	ACCC	0.019527	ATCC	0.000752
AACG	0.000843	AGCG	0.002202	ACCG	0.000315	ATCG	0.000048
AACT	0.000121	AGCT	0.000048	ACCT	0.000752	ATCT	0.001546
AAGA	0.028625	AGGA	0.005985	ACGA	0.000702	ATGA	0.000234
AAGC	0.000702	AGGC	0.000755	ACGC	0.001837	ATGC	0.000116
AAGG	0.005985	AGGG	0.032738	ACGG	0.000755	ATGG	0.000252
AAGT	0.000234	AGGT	0.000252	ACGT	0.000116	ATGT	0.000535
AATA	0.003025	AGTA	0.000281	ACTA	0.000121	ATTA	0.000154
AATC	0.000121	AGTC	0.000048	ACTC	0.001781	ATTC	0.000517
AATG	0.000281	AGTG	0.000734	ACTG	0.000048	ATTG	0.000073
AATT	0.000154	AGTT	0.000073	ACTT	0.000517	ATTT	0.004711

Patterns starting with C:

Pattern	Prob.	Pattern	Prob.	Pattern	Prob.	Pattern	Prob.
CAAA	0.018317	CGAA	0.001490	CCAA	0.000628	CTAA	0.000166
CAAC	0.000628	CGAC	0.000210	CCAC	0.009592	CTAC	0.000415
CAAG	0.001490	CGAG	0.002878	CCAG	0.000210	CTAG	0.000048
CAAT	0.000166	CGAT	0.000048	CCAT	0.000415	CTAT	0.001214
CACA	0.005277	CGCA	0.000669	CCCA	0.004524	CTCA	0.000375
CACC	0.004524	CGCC	0.002262	CCCC	0.167489	CTCC	0.005866
CACG	0.000669	CGCG	0.002304	CCCG	0.002262	CTCG	0.000188
CACT	0.000375	CGCT	0.000188	CCCT	0.005866	CTCT	0.007452
CAGA	0.003304	CGGA	0.001065	CCGA	0.000210	CTGA	0.000048
CAGC	0.000210	CGGC	0.000209	CCGC	0.004796	CTGC	0.000208
CAGG	0.001065	CGGG	0.006655	CCGG	0.000209	CTGG	0.000059
CAGT	0.000048	CGGT	0.000059	CCGT	0.000208	CTGT	0.000607
CATA	0.000959	CGTA	0.000120	CCTA	0.000360	CTTA	0.000404
CATC	0.000360	CGTC	0.000180	CCTC	0.011625	CTTC	0.001716
CATG	0.000120	CGTG	0.000420	CCTG	0.000180	CTTG	0.000202
CATT	0.000404	CGTT	0.000202	CCTT	0.001716	CTTT	0.013873

Patterns starting with G:

Pattern	Prob.	Pattern	Prob.	Pattern	Prob.	Pattern	Prob.
GAAA	0.045565	GGAA	0.005060	GCAA	0.001004	GTAA	0.000335
GAAC	0.001004	GGAC	0.000453	GCAC	0.001837	GTAC	0.000116
GAAG	0.005060	GGAG	0.017648	GCAG	0.000453	GTAG	0.000151
GAAT	0.000335	GGAT	0.000151	GCAT	0.000116	GTAT	0.000535
GACA	0.002514	GGCA	0.000532	GCCA	0.000315	GTCA	0.000048
GACC	0.000315	GGCC	0.000194	GCCC	0.009764	GTCC	0.000376
GACG	0.000532	GGCG	0.002904	GCCG	0.000194	GTCC	0.000036
GACT	0.000048	GGCT	0.000036	GCCT	0.000376	GTCT	0.000773
GAGA	0.014437	GGGA	0.008240	GCGA	0.000476	GTGA	0.000159
GAGC	0.000476	GGGC	0.001251	GCGC	0.001823	GTGC	0.000117
GAGG	0.008240	GGGG	0.056794	GCGG	0.001251	GTGG	0.000417
GAGT	0.000159	GGGT	0.000417	GCGT	0.000117	GTGT	0.000530
GATA	0.000838	GGTA	0.000177	GCTA	0.000048	GTTA	0.000073
GATC	0.000048	GGTC	0.000036	GCTC	0.000891	GTTT	0.000258
GATG	0.000177	GGTG	0.000968	GCTG	0.000036	GTTG	0.000040
GATT	0.000073	GGTT	0.000040	GCTT	0.000258	GTTT	0.002355

Patterns starting with T:

Pattern	Prob.	Pattern	Prob.	Pattern	Prob.	Pattern	Prob.
TAAA	0.006106	TGAA	0.000497	TCAA	0.000166	TTAA	0.000099
TAAC	0.000166	TGAC	0.000048	TCAC	0.001389	TTAC	0.000240
TAAG	0.000497	TGAG	0.000959	TCAG	0.000048	TTAG	0.000038
TAAT	0.000099	TGAT	0.000038	TCAT	0.000240	TTAT	0.002009
TACA	0.000959	TGCA	0.000120	TCCA	0.000548	TTCA	0.000215
TACC	0.000548	TGCC	0.000274	TCCC	0.019456	TTCC	0.001275
TACG	0.000120	TGCG	0.000420	TCCG	0.000274	TTCG	0.000108
TACT	0.000215	TGCT	0.000108	TCCT	0.001275	TTCT	0.006924
TAGA	0.001101	TGGA	0.000355	TCGA	0.000048	TTGA	0.000038
TAGC	0.000048	TGGC	0.000059	TCGC	0.000694	TTGC	0.000120
TAGG	0.000355	TGGG	0.002218	TCGG	0.000059	TTGG	0.000030
TAGT	0.000038	TGGT	0.000030	TCGT	0.000120	TTGT	0.001005
TATA	0.001119	TGTA	0.000143	TCTA	0.000231	TTTA	0.000893
TATC	0.000231	TGTC	0.000116	TCTC	0.004935	TTTC	0.003240
TATG	0.000143	TGTG	0.000488	TCTG	0.000116	TTTG	0.000447
TATT	0.000893	TGTT	0.000447	TCTT	0.003240	TTTT	0.031522

Bibliography

- Edwards, A. W. F. 2009. Statistical methods for evolutionary trees. *Genetics* 183:5–12.
- Felsenstein, J. 1981. Evolutionary trees from DNA sequences: A maximum likelihood approach. *Journal of Molecular Evolution* 17:368–376.
- Fisher, R. A. 1950. Gene frequencies in a cline determined by selection and diffusion. *Biometrics* 6:353–361.
- Gallager, R. G. 1962. Low-density parity-check codes. *IRE Trans. Inform. Theory* 8:21–28.
- Gallager, R. G. 1963. Low-density parity check codes. MIT Press, Cambridge, MA.
- Gelfand, A. E. and A. F. M. Smith. 1990. Sampling-based approaches to calculating marginal densities. *J. Amer. Statist. Assoc.* 85:398–409.
- Geman, S. and D. Geman. 1984. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 6:721–741.
- Hasegawa, M., H. Kishino, and T. Yano. 1985. Dating the human-ape splitting by a molecular clock of mitochondrial DNA. *Journal of Molecular Evolution* 22:160–174.
- Hasegawa, M., T. Yano, and H. Kishino. 1984. A new molecular clock of mitochondrial DNA and the evolution of Hominoids. *Proc. Japan Acad. Ser. B* 60:95–98.
- Hastings, W. K. 1970. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika* 57:97–109.
- Höhna, S., M. J. Landis, T. A. Heath, B. Boussau, N. Lartillot, B. R. Moore, J. P. Huelsenbeck, and F. Ronquist. 2016. Revbayes: Bayesian phylogenetic inference using graphical models and an interactive model-specification language. *Systematic Biology* 65:726–736.
- Kendall, D. G. 1948. On the generalized “birth-death” process. *Annual Review of Mathematical Statistics* 19:1–15.
- Metropolis, N., A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller. 1953. Equation of state calculations by fast computing machines. *Journal of Chemical Physics* 21:1087–1092.
- Nee, S., R. M. May, and P. H. Harvey. 1994. The reconstructed evolutionary process. *Philosophical Transactions of the Royal Society of London B* 344:305–311.

- Nielsen, R. 2002. Mapping mutations on phylogenies. *Systematic Biology* 51:729–739.
- Park, S. K. and K. W. Miller. 1988. Random number generators: good ones are hard to find. *Communications of the ACM* 31:1192–1201.
- Schröder, E. 1870. Vier combinatorische probleme. *Zeitschrift für Mathematik und Physik* 15:361–376.
- Tavaré, S. 1986. Some probabilistic and statistical problems on the analysis of DNA sequences. *Lectures in Mathematics in the Life Sciences* 17:57–86.
- Thompson, E. A. 1975. *Human Evolutionary Trees*. Cambridge University Press, Cambridge, England.
- Tuffley, C. and M. Steel. 1997. Links between maximum likelihood and maximum parsimony under a simple model of site substitution. *Bulletin of Mathematical Biology* 59:581–607.