Tutorial 1: RevBayes Basics

1. Installing RevBayes

Open up a terminal window on your Macintosh with the "Terminal" app (you can find it in /Applications/Utilities/ - best drag it onto the dock). Create a directory called "revbayes" using the following command:

```
$ mkdir revbayes
```

Now go to the DropBox folder at
https://www.dropbox.com/sh/4w0rjkt02q2xf3n/AAC4iifv_FptFh-D_AoObUgEa?dl=0

Using Finder or the terminal, find the folder "revbayes" on your computer (it will be in your home directory) and copy the DropBox file "rb" and the DropBox folders "lib" (the folder itself and all of its contents) and "help" into your folder "revbayes".

Use the terminal to navigate into the directory revbayes, by typing

```
$ cd ~/revbayes/
```

Check if all the files are there by typing

```
$ ls -l
```

Now to make sure that you can actually execute the "rb" program, type

```
$ chmod 777 rb
```

To make it possible for your system to find the executable "rb" and the boost libraries in "lib", we need to tell it where they are. Do that by editing (or creating) your .bash_profile file *in your home directory* using the text editor nano:

```
$ nano .bash_profile
```

After the editor window appears, add these two lines to the top of the file. The lines must be printed *exactly* as they appear here except that there should be no line break in the second line. All words are case sensitive!

```
export PATH=/Volumes/Data/Users/workshop2014/revbayes:$PATH

export
DYLD_LIBRARY_PATH=/Volumes/Data/Users/workshop2014/revbayes/lib:$DYLD_L
IBRARY_PATH
```

Save the file and close nano by pressing "ctrl + O", "return", and then "ctrl + X".

Test that everything worked by closing "Terminal" and restarting it. Typing

```
$ rb
```

should now start RevBayes and print the following welcome message (or something like it):

```
RevBayes version 1.0.0 beta (2014-11-15)

RevBayes provides an interactive environment for Bayesian
phylogenetic modeling and inference. It is based on probabi-
listic graphical model concepts and uses its own interpreted
language, Rev. For more info, visit www.RevBayes.com.

RevBayes is free software released under the GPL license,
version 3. Type 'license()' for details.

RevBayes is a collaborative project with many contributors.
Type 'contributors()' for more information. Type 'citation()'
for information on how to cite RevBayes, and 'contacts()' for
contact details of core contributors.

For help on using RevBayes, type 'help()'. To quit RevBayes
type 'quit()' or 'q()'.
```

2. Getting Started

2.1. RevBaye as a Calculator

You can use RevBayes as a simple calculator. Just type in the expression and press enter to evaluate it:

```
> 1 + 3
```

After you press "return", RevBayes should print the answer:

```
    4
```

RevBayes ignores spaces, so you can use spaces between numbers and operators or you can leave them out, the result should be the same.

You can store intermediate results by assigning them to variables using the left arrow operator, consisting of the "<" (less than) and "-" (dash or minus) characters on your keyboard. You can print the value of any variable by simply typing the name of that

variable followed by "return". For instance, to calculate 4 * (3 + 2), we could use the following lines:

```
> a <- 1 + 3
> b <- 4 * ( 3 + 2 )
> b
```

Note that the last line causes the value of "b" to be printed.

As you define variables in this way, they get added to your workspace. To list the current content of your workspace, use the "ls" function. All function names need to be followed by parentheses containing the parameters of the function. If you do not wish to pass any arguments to the function, the parentheses would not contain anything, but they are still required. Thus, to call "ls" without parameters, type:

```
> ls()
```

To clear the workspace, simply use the "clear" function, which does not take any arguments. You can redefine any variable at any time by simply assigning a new value to it. Note that RevBayes is case-sensitive, so a variable called "myvariable" is different from one called "MyVariable".

Exercise 1. Calculate the number of days until New Year's Day (January 1) 2020 using RevBayes.

Exercise 2. An organism reproduces at an exponential rate *r* for each generation, resulting in the growth equation

$$N = N_0 e^{rt}$$

where N is the number of individuals after t generations, and N_0 is the original number of individuals. Use RevBayes to calculate the number of individuals after $t = 10$ generations when $r = 0.2$ and $N_0 = 5$. Hint: Use the "exp(x)" function to get the value of e raised to some power x.

2.2. Type and Structure of Variables

The language used by RevBayes has many types. However, RevBayes assigns types to variables and literal constants implicitly from the statements you use to create them, so you usually do not have to specify the type yourself.

The seven basic types of scalar variables are:

Type	Description
String	String (enter as quoted string)
Bool	Boolean (either TRUE or FALSE)
Integer	Whole number
Natural	Non-negative whole number (0, 1, ...)
Real	Real number
RealPos	Positive real number
Probability	Positive real number between 0.0 and 1.0

To see the type of a variable or constant, you can use the ”type” function. To see the type of the constant ”2.0”, use:

```
> type( 2.0 )
```

To get more detailed information about the structure of the variable or constant, use the “str” function. For instance, to see the structure of a variable called “myVar”, which has the value “2.0”, use

```
> myVar <- 2.0
> str( myVar )
```

The output will look like this:

```
_variable     = myVar
_RevType      = RealPos
_RevTypeSpec  = [ RealPos, Real, RevObject ]
_value        = 2
_dagType      = Constant DAG node
_children     = [  ]
.methods = void function ()
```

Here, you can see that “myVar” is of type “Probability”. The “RevTypeSpec” attribute gives you the inheritance of the type. It tells you that “Probability” is a type derived from “RealPos”, which is derived from “Real” etc. That is, “myVar” can be used in all situations where you expect a “RealPos” or a “Real” value, because the type “Probability” is a specialized form of these.

The output also tells you the type of DAG node represented by the variable, any children or parents it might have (more about this soon). Finally, you will get a list of the member methods, with their call signatures, and member variables, if any. Here, you can see that the variable “myVar” has a single member method called “methods”, which does not take any arguments, and returns nothing (void). To call it and see what it does, you use the “dot” operator after the variable name, and then the name of the method:

```
> myVar.methods()
methods = void function ()
```

The member method "methods" just prints a list of the member methods, that is, part of the output of the "str" function. There are often many different ways of accomplishing the same thing in RevBayes.

Exercise 3. Find out the inheritance hierarchy of the seven scalar types using the "str" function on suitably specified constants or variables Hint: In the example above, RevBayes figured out that "2.0" is a "RealPos" value, because it is outside of the range of a "Probability", but still inside the range of a "RealPos" value.

2.3. Getting Help on Functions

Calling a function is the standard way of making RevBayes calculate or perform something. It is often helpful to know the arguments and return type of a particular function (its "call signature"). You can obtain this information by simply typing the name of the function. For instance, to get the call signature of the "log" function, type:

```
> log
```

and you will get

```
   Real function (RealPos x, RealPos base)
```

This means that the "log" function takes a real positive argument "x" and a real positive argument "base", and returns a real value. If you specify the labels, you can pass in the arguments in any order. If you do not specify them, you have to give the arguments in the expected order, that is, first the number and then the base of the logarithm. For instance, to calculate the base-10 logarithm of 5.0, we could use

```
> log( 5.0, 10.0 )
```

or, equivalently, using argument labels to swap the order of the values passed in:

```
> log( base = 10.0, x = 5.0 )
```

More extensive help on a function is also available (sometimes) by typing a question mark followed by the name of the function. For instance:

```
> ? log
```

should print the help information for the log function. The help system is still incomplete, so RevBayes may complain that it cannot find any help information for the specified function, or it may display information from a placeholder file, which is not very useful.

A function we will use later is called “mcmc”, for which there is a detailed description available. You can examine the structure of this file by typing

```
> ? mcmc
```

Exercise 4. What is the return type of the “exp” function? Why is it different from the return type of the “log” function?

2.4. Working with Vectors

Vectors are different from scalars in Rev, in contrast to R. The easiest way to create a vector in Rev is to assign to individual elements:

```
> a[1] <- 1
> a[2] <- 2
> a
   [ 1, 2 ]
> type(a)
   Natural[]
```

Note that the type of vector of natural numbers is ”Natural[]”, and that the vector is printed enclosed in square brackets and elements separated by comma.

It is also possible to create the entire vector at once using either of the constructs:

```
> a <- [ 1, 2 ]
> a <- v( 1, 2 )
```

Finally, we can create a vector with a loop, which is structured similarly to R:

```
> for ( i in 1:2 )
+    a[i] <- i
```

Note the ”+” sign indicating the need for additional input.

Exercise 5. Create a vector mixing values of different type: natural, negative real number, real number, integer. What is the type of the vector?

2.5. Command History and Tab Completion

RevBayes keeps a history of your previous commands, so it is easy to retrieve previous commands and modify them. This is often very helpful when working interactively with the environment.

For instance, there is a function called ”rnorm” that generates a vector of random values from a normal distribution (as in R). Each time you call the function, it generates a new vector of values. Assume we want to see this in action, by repeatedly drawing ten

numbers from a standard normal (mean 0.0 and standard deviation 1.0). To draw the first set of variables, use

```
> rnorm( 10, 0.0, 1.0 )
```

You should get a vector of ten draws from the distribution. You can get another draw of ten values by typing in the same statement again. However, a much quicker way of doing the same thing is to simply press the up-arrow key on your keyboard, which will bring back the previous command. You can step back in that command using the backspace key or the left-arrow key and edit it if needed. Once you are happy with the line, press return to execute it. If you have entered several commands, you can step backwards in the command history by pressing the up-arrow as many times as required to retrieve the command you are interested in.

Exercise 6. First, generate 10 random draws from the standard normal as described above. Then bring back the line, edit it, and generate 100 random draws instead. Then bring back the first line and edit it to generate 10 draws from a normal distribution with mean 1.0 and standard deviation 10.0.

Another useful feature is tab completion. You can use it in various contexts to complete part of a statement. If there are several completion possibilities, repeatedly pressing the tab key will step through the possibilities. For instance, let us assume that you are too lazy to type the entire name of the “rnorm” function. Then you can simply type “rno” (without spaces) and then press the tab key to complete the name of the function (there should be only one function name starting with “rno”.

Exercise 7. Use tab completion to find out how many function names start with “r”. You can confirm that you get the right answer by printing all functions in the work space using “ls(all=true)”, and then count the number of function names starting with “r” in the output.

2.6. Defining a Simple Model

Now it is time to define our first models. First clear everything from your workspace using

```
> clear()
```

Now let us create a random variable ”x” drawn from a standard normal. For this, we use a stochastic assignment (tilde assignment) with a distribution constructor function on the right-hand side. The distribution constructor functions all have names that start with ”dn”, and the normal distribution has a constructor function named ”dnNorm”. So, we create the random variable ”x” using the following statement:

```
> x ~ dnNorm( 0.0, 1.0 )
```

Look at the value of ”x” by typing ”x”. It should be a real value. You can also verify that the arguments of the ”dnNorm” constructor function are the mean and the standard deviation by typing ”dnNorm”.

Now look at the structure of “x” by typing:

```
> str(x)
```

You should get something like

```
   _variable     = x
   _RevType      = Real
   _RevTypeSpec  = [ Real, RevObject ]
   _value        = -1.30911
   _dagType      = Stochastic DAG node
   _clamped      = FALSE
   _lnProb       = -1.77582
   _parents      = [ <0x7fe13b114920>, <0x7fe13b114c90> ]
   _children     = [  ]
   .clamp = void function (Real x)
   .lnProbability = Real function ()
   .methods = void function ()
   .probability = RealPos function ()
   .redraw = void function ()
   .setValue = void function (Real x)
   .unclamp = void function ()
```

The type of “x” is “Real”, but it is a stochastic node and not a constant node. It has two parents in the workspace model DAG, consisting of the unnamed literal constants “0.0” (the mean) and “1.0” (the standard deviation). Because they are unnamed, RevBayes prints their memory addresses instead of printing names. It may be convenient to name them; so let us do that by assigning them to constant variables before using them.

```
> clear()
> mean <- 0.0
> sd <- 1.0
> x ~ dnNorm( mean, sd )
> str(x)
```

```
   _variable     = x
   _RevType      = Real
   _RevTypeSpec  = [ Real, RevObject ]
   _value        = 0.160513
   _dagType      = Stochastic DAG node
   _clamped      = FALSE
   _lnProb       = -0.931821
```

```
_parents      = [ sd, mean ]
_children     = [  ]
.clamp = void function (Real x)
.lnProbability = Real function ()
.methods = void function ()
.probability = RealPos function ()
.redraw = void function ()
.setValue = void function (Real x)
.unclamp = void function ()
```

Now the parents are named (the order of the parents is not necessarily the same as the order of the arguments to the "dnNorm" constructor). Focusing on the other attributes, we see that the value has changed. The variable is unclamped ("_clamped = FALSE"). A number of member methods have been added to the variable. Among other things, they allow us to see the probability density or log probability density of the current value. We can also clamp the node to an observed value or unclamp it. The "redraw" function draws a new value of the variable. Finally, "setValue" allows us to hard-set the value without clamping the node.

Exercise 8. Redraw the value of "x" repeatedly and check that the probability density function appears to have the correct bell shape centered on 0.0.

We have constructed a small model now in our workspace. To see the structure of the model, create the model using the "model" function. The "model" function just creates a copy of the DAG in the workspace, using any named model variable as its argument. We do not save the model, but just print it by using the command:

```
> model(x)
```

The output should look something like:

```
Model with 3 nodes
==================

sd :
_value        = 1
_dagType      = Constant DAG node
_children     = [ x ]

mean :
_value        = 0
_dagType      = Constant DAG node
_children     = [ x ]

x :
```

```
_value          = 0.160513
_dagType        = Stochastic DAG node
_clamped        = FALSE
_lnProb         = -0.931821
_parents        = [ sd, mean ]
_children       = [  ]
```

To save the model for future use, we assign it to a variable. This is neither a constant nor a stochastic node but a so-called workspace variable. Therefore, RevBayes uses the assignment operator "=" instead of the left arrow or tilde operators. Create a copy of the model called "myModel", use something like

```
> myModel = model(x)
```

Instead of "x", you can also use "mean" or "sd".

If we modify the variables in the workspace, the copy of the model in "myModel" is not affected. Check this by replacing the constant node "mean" with the constant value 2.0.

```
> mean <- 2.0
> myModel2 = model( mean )
```

Now verify that the two models called "myModel" and "myModel2" are different.

Exercise 9. Start from scratch and specify a hierarchical normal model, where the mean and standard deviation are drawn from suitable probability distributions instead of being constant. For instance, you can use a uniform distribution ("dnUnif") for the mean and an exponential distribution ("dnExp") for the standard deviation. Furthermore, make sure that the random variable "x" is a vector of Real values of length 10 instead of being a single value. Hint: Use a loop to make a stochastic assignment (tilde assignment) to each of the elements of the vector "x", just like we created a vector using a loop with left-arrow assignments above.

2.7. Estimating Model Parameters with Bayesian MCMC Inference

Let us now use MCMC to simulate values drawn from a standard normal distribution. To do this, we first need to create a suitable model:

```
> clear()
> x ~ dnNorm( 0.0, 1.0 )
> myModel = model( x )
```

Now we assign a sliding window move to the random variable "x" using the appropriate constructor. All move constructors have names starting with "mv", and the sliding window move constructor is "mvSlide". We store the move in a move vector that we call "moves" for later use:

```
> moves[1] = mvSlide( x )
```

If you wish to check default tuning parameter value, weight etc for the move, just type "mvSlide". We go with the default values here by not passing any alternative values into the constructor function.

To monitor the value of "x" during the run, we create a screen monitor for "x", and save it into a "monitors" vector. We also add a file monitor that prints the sampled values of "x" to a file called "out.txt". All monitor constructor functions start with "mn".

```
> monitors[1] = mnScreen( x )
> monitors[2] = mnFile( x, filename = "output.txt" )
```

Now we create an mcmc object called "myMcmc" with the "mcmc" constructor function, and finally we run it for 100 generations using the "run" member method:

```
> myMcmc = mcmc( myModel, monitors, moves )
> myMcmc.run( 100 )
```

Now you should see the sampled values flip by on screen. After the analysis has completed, you can look at the performance of moves using the member method "operatorSummary". You can also obtain a quick summary of the sampled values using the "readTrace" function. For more sophisticated post-hoc analyses, use software like R to read in the content of the file produced by the file monitor.

```
> myMcmc.operatorSummary()
> readTrace( "output.txt" )
```

Exercise 10. Use the model outlined in Exercise 9. Generate 10 random draws from a normal distribution with known mean and standard deviation. Clamp the normal random variables in your model using those values. Finally, set up an MCMC analysis and see if you can estimate the mean and the standard deviation from those data. Remember to set up a file monitor (and possibly a screen monitor) for all the parameters you want to sample. Alternatively, use a model monitor ("mnModel"), which samples all model parammeters to a named file.

Tip: If you want to reuse a model over and over again, you can save it to a Rev script file (a text file); we propose you use the file ending ".Rev" to signal that the file contains Rev code. To read in the commands in the file, simply use the "source" command with the name of the script file as its argument.

2.8. Deterministic Nodes

Many models use variable transformations. Such dynamic transformations are accomplished using deterministic nodes in a graphical model. They are created using the ":=" assignment operator in Rev. In contrast to a constant node, the value of a deterministic node is updated when the value of its arguments change. To see this, play with this example:

```
> a <- 1
> b <- exp(a)
> c := exp(a)
> b
   2.71828
> c
   2.71828
> a <- 2
> b
   2.71828
> c
   7.38906
```

Let us use this in our normal model. Some statisticians prefer to parameterize the normal distribution by the mean and the precision instead of using the mean and the standard deviation. The precision is one over the variance of the distribution, and the variance is the square of the standard deviation.

Assume that, following the BUGS manual, we want to put a Gamma(0.001,0.001) prior on the precision parameter. Then a random variable "x" drawn from a hierarchical normal distribution of mean "mean" and precision "prec" could be specified as follows. Note that we insert a deterministic variable transformation between the "prec" stochastic node and the variable "sd" used as a parameter in the normal distribution.

```
> clear()
> prec ~ dnGamma( 1.0E-3, 1.0E-3 )
> mean ~ dnNorm( 0.0, 100.0 )
> sd := sqrt( 1.0 / prec )
> x ~ dnNorm( mean, sd )
```

Exercise 11. Repeat Exercise 10 using the suggested prior on the precision of the normal distribution. Are the estimates different?

2.9. Writing Functions

You can easily write your own functions in Rev. Assume you want a function that finds the square of a number. You define the function using the following syntax:

```
> function RealPos foo( Real x ) { x * x }
```

In principle, you can define a function without specifying the types of the argument(s) or the return type of the function, but it is not recommended because the type then defaults to RevObject and the behavior might be unexpected (or erroneous because of bugs). In particular, such a function is almost surely going to cause problems if you try to include it in your model.

Now calculate the square of some interesting numbers by using the function:

```
> foo(0.5)
   0.25
> foo(exp(1))
   7.38906
```

Exercise 12. Define a function that takes the square root of the inverse of a number (the function we needed for the deterministic node in Exercise 11). Repeat Exercise 11 using this function instead of using an explicit expression.