Statistical Inference using RevBayes

Basic introduction to Rev & MCMC

1 Overview

`RevBayes` has as its central idea that any statistical model, for example a phylogenetic model, is composed of smaller parts that can be decomposed and put back together in a modular fashion. This comes from considering (phylogenetic) models as *probabilistic graphical models*, which lends flexibility and enhances the capabilities of the program. Users interact with `RevBayes` via an interactive shell. Users communicate commands using a language specifically designed for `RevBayes`, called `Rev`; an R-like language (complete with control statements, user-defined functions, and loops) that enables the user to build up (phylogenetic) models from simple parts (random variables, variable/parameter transformations, models, and constants of different sorts).

This tutorial demonstrates the basic syntactical features of `RevBayes` and `Rev` and shows how to set up and perform an analysis on "toy" statistical models for linear regression. This tutorial focuses on explaining probabilistic graphical models and the language `Rev`. A good reference for probabilistic graphical models for Bayesian phylogenetic inference is given in Höhna et al. (2014). The statistical examples are borrowed from a fourth year statistics course taught in the fall term 2011 at Stockholm University.

1.1 Probabilistic Graphical Models

`RevBayes` uses *probabilistic graphical models* for model specification, visualization, and implementation (Höhna et al. 2014). Graphical models are frequently used in machine learning and statistics to conceptually represent the conditional dependence structure of complex statistical models with many parameters (Gilks et al. 1994; Lunn et al. 2000; Jordan 2004; Koller and Friedman 2009; Lunn et al. 2009). The graphical model framework allows for flexible model specification and implementation and reduces redundant code. This framework provides a set of symbols for depicting a *directed acyclic graph* (DAG). Höhna et al. (2014) described the use of probabilistic graphical models for phylogenetics. The different nodes and components of a phylogenetic graphical model are shown in Figure 1 (Fig. 1 from Höhna et al. 2014).

To represent the DAG, nodes are connected with arrows indicating dependency. A simple, albeit abstract, graphical model is shown in Figure 2. In this model, we observe a set of states for parameter x. We assume that the values of x are samples from a lognormal distribution with a location parameter (log mean) μ and a standard deviation σ. It is more straightforward to model our uncertainty in the expectation of a lognormal distribution, rather than μ, thus we place a gamma distribution on the mean M. This gamma hyperprior has two parameters that we specify with fixed values (constant nodes): the shape α and rate β. The variable M is a stochastic node with this prior density. The standard deviation, σ, is also a stochastic node with an exponential prior density with rate parameter λ. For any value of M and any value of σ we can compute the deterministic variable μ using the formula $\mu = \ln(M) - \frac{\sigma^2}{2}$. This formula is known from using simple algebra on the equation for the mean of any lognormal distribution. With this model structure, we can then calculate the probability of the data conditional on the model and parameter values (the likelihood): $\mathbb{P}(\boldsymbol{x} \mid \mu, \sigma)$. Next we can get the posterior probability using Bayes' theorem:

$$\mathbb{P}(M, \sigma \mid \boldsymbol{x}, \alpha, \beta, \lambda) = \frac{\mathbb{P}(\boldsymbol{x} \mid \mu, \sigma)\mathbb{P}(M \mid \alpha, \beta)\mathbb{P}(\sigma \mid \lambda)}{\mathbb{P}(\boldsymbol{x})}.$$

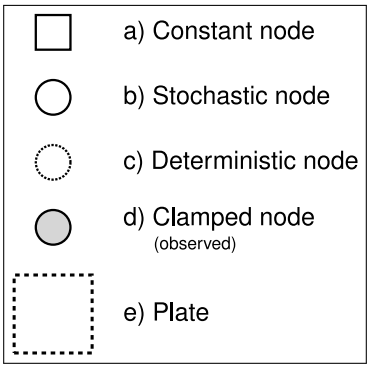

Figure 1: The symbols for a visual representation of a graphical model. a) Solid squares represent constant nodes, which specify fixed-valued variables. b) Stochastic nodes are represented by solid circles. These variables correspond to random variables and may depend on other variables. c) Deterministic nodes (dotted circles) indicate variables that are determined by a specific function applied to another variable. They can be thought of as variable transformations. d) Observed states are placed in clamped stochastic nodes, represented by gray-shaded circles. e) Replication over a set of variables is indicated by enclosing the replicated nodes in a plate (dashed rectangle). [Partially reproduced from Fig. 1 in Höhna et al. (2014).]

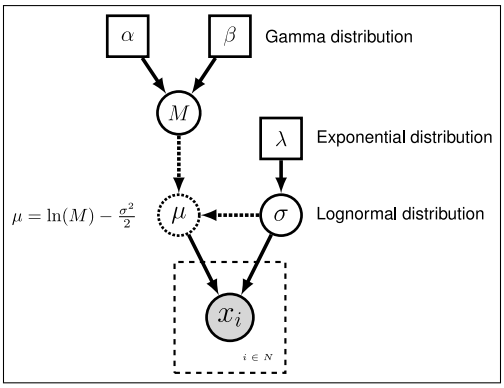

Figure 2: Graphical model representation of a simple lognormal model. A total of N observations of variable x are observed and occupy a clamped node. This parameter is log-normally distributed with parameters μ and σ (log mean and standard deviation, respectively). The parameter μ is a deterministic node that is calculated from the stochastic nodes M (the mean of the distribution) and σ. Dotted arrows indicate deterministic functions and are used to connect deterministic nodes to their parent variables. A gamma distribution is applied as a hyper prior on M with constant nodes for the shape α and rate β. The stochastic variable σ is exponentially distributed with fixed value for the rate λ.

Rev: The RevBayes Language

In RevBayes models and analyses are specified using an interpreted language called Rev. Rev bears similarities to the compiled language in WinBUGS and the interpreted R language. Setting up and executing a statistical analysis in RevBayes requires the user to specify all of the parameters of their model and the type of analysis (e.g., , an MCMC run). By using an interpreted language, RevBayes enables the practitioner to build complex, hierarchical models and to check the current states of variables while building the model. This will be very useful in the beginning. Later on you, when you run very complex analyses, you may want to write Rev-scripts.

Differently to R and BUGS, Rev is a strongly but implicitly typed language. It is implicitly typed, and thus similar to Python, because you do not need to provide the type of a variable (which you need to in languages

such as C++ and Java). We do implicit typing to help users who do not know about the actual types of the variables. However, strongly typed means that every variable has a type and arguments of functions need to match the required types. The strong type requirements ensures that you build meaningful model graphs. For example, the variance parameter of a normal distribution needs to be a positive number, and thus you can only use variables that are positive real numbers. `RevBayes` does automatic type conversion.

Specifying Models

Table 1: `Rev` assignment operators, clamp function, and plate/loop syntax.

Operator	Variable
`<-`	constant variable
`~`	stochastic variable
`:=`	deterministic variable
`node.clamp(data)`	clamped variable
`for(i in 1:N){...}`	plate

The variables/parameters of a statistical model are created using different operators in `Rev` (Table 1). In Figure 3, the `Rev` syntax for creating the model in Figure 2 is provided. Because `Rev` is an interpreted language, it is important to consider the order in which you specify your variables (cf. BUGS where the order is not important). Thus, typically the first variables that are instantiated are *constant variables*. Constant variables require you to assign a fixed value using the `<-` operator. Stochastic variables are initialized using the `~` operator followed by the constructor function for a distribution. In `Rev`, the naming convention for distributions is `dn*`, where `*` is a wildcard representing the name of the distribution. Each distribution function requires hyper-parameters passed in as arguments. This is effectively linking nodes using arrows in the graphical model. The following code snippet creates a stochastic variable called `M` which is assigned a gamma-distributed hyperprior, with shape `alpha` and rate `beta`:

```
alpha <- 2.0
beta <- 4.0
M ~ dnGamma(alpha, beta)
```

The flexibility gained from the graphical model framework and the interpreted language allows you to easily change a model by swapping components. For example, if you decide that a bimodal lognormal distribution is a better representation of your uncertainty in `M`, then you can simply change the distribution associated with `M` (after initializing the bimodal lognormal hyperparameters):

```
mean_1 <- 0.5
mean_2 <- 2.0
sd_1 <- 1.0
sd_2 <- 1.0
weight <- 0.5
M ~ dnBimodalLnorm(mean_1, mean_2, sd_1, sd_2, weight)
```

`Rev` does allow you to specify constant-variable values in the distribution constructor function, therefore this also works:

```
M ~ dnBimodalLnorm(0.5, 2.0, 1.0, 1.0, 0.5)
```

Both ways to specify priors are equivalent. The only difference is that one code may be more readable than the other.

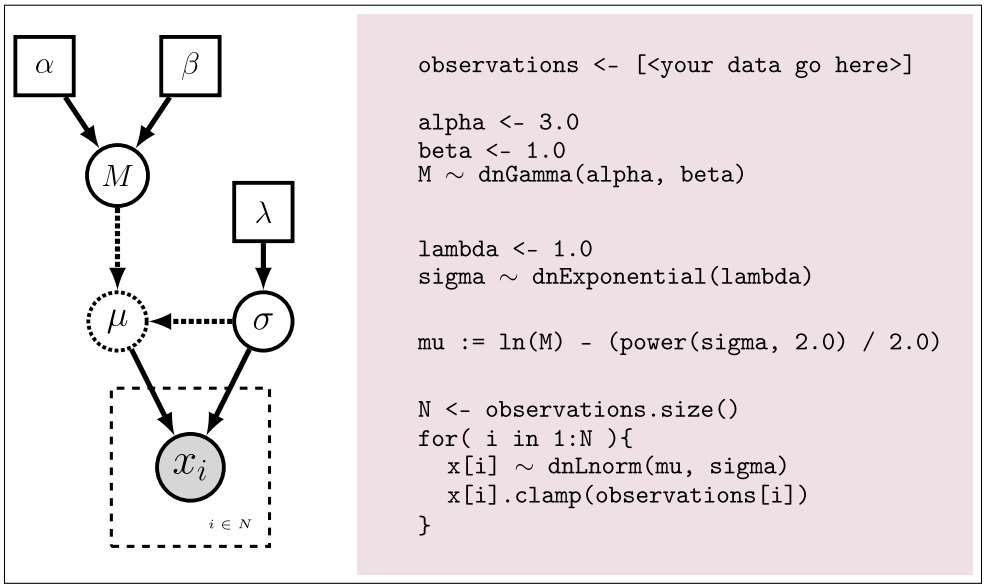

Figure 3: Specifying a model with `Rev`. The graphical model of the observed parameter x is shown on the left. In this example, x is log-normally distributed with a location parameter of μ and a standard deviation of σ, thus $x \sim \text{Lognormal}(\mu, \sigma)$. The expected value of x (or mean) is equal to M: $\mathbb{E}(x) = M$. In this model, M and σ are random variables and each are assigned hyperpriors. We assume that the mean is drawn from a gamma distribution with shape parameter α and rate parameter β: $M \sim \text{Gamma}(\alpha, \beta)$. The standard deviation of the lognormal distribution is assigned an exponential hyperprior with rate λ: $\sigma \sim \text{Exponential}(\lambda)$. Since we are conditioning our model on the *expectation*, we must compute the location parameter (μ) to calculate the probability of our model. Thus, μ is a deterministic node that is the result of the function* executed on M and σ: $\mu = \ln(M) - \frac{\sigma^2}{2}$. Since we observe values of x, we *clamp* this node.

Deterministic variables are parameter transformations and initialized using the `:=` operator followed by the function or formula for calculating the value. Previously we created a variable for the expectation of the lognormal distribution. Now, if you have an exponentially distributed stochastic variable σ, you can create a deterministic variable for the mean μ:

```
lambda <- 1.0
sigma ~ dnExponential(lambda)
mu := ln(M) - (sigma^2)/2.0
```

Replication over lists of variables as a plate object is specified using **`for`** loops. A for-loop is an iterator statement that performs a function a given number of times. In `Rev` you can use this syntax to create a vector of 7 stochastic variables, each drawn from a lognormal distribution:

```
for( i in 1:7 ) {
  x[i] ~ dnLognormal(mu, sigma)
}
```

The **for** loop executes the statement **x[i] dnLognormal(mu, sigma)** for different values of i repeatedly, where i takes the values 1 to 7. Thus, we created a vector x of seven variables, each being independent and identically distributed (i.i.d.).

A clamped node/variable has observed data attached to it. Thus, you must first read in or input the data, then clamp it to a stochastic variable. In Figure 3 the observations are assigned and clamped to the stochastic variables. If we observed 7 values for **x** we would create 7 clamped variables:

```
observations <- [0.20, 0.21, 0.03, 0.40, 0.65, 0.87, 0.22]
N <- observations.size()
for( i in 1:N ){
  x[i].clamp(observations[i])
}
```

You may notice that the value of x has now changed and is equal to the observations.

Getting help in RevBayes

RevBayes provides an elaborate help system. Most of the help is found online on our website http://www.RevBayes.com. Within RevBayes you can display the help for a function, distribution or any other type using the **?** symbol followed by the command you want help for:

```
?dnNorm
?mcmc
?mcmc.run
```

Additionally, RevBayes will print the correct usage of a function if you only type in its name and hit return:

```
mcmc
   MCMC function (Model model, Monitor[] monitors, Move[] moves, String moveschedule = "
    sequential" | "random" | "single"
```

Continue on to the next page to start the tutorial...

2 Tutorial: Basic Rev Commands

2.1 Introduction

The first section of this tutorial involves

1. Creating different types of variables.
2. Learning about functions.

All of the files for this analysis are provided for you and you can run these without significant effort using the **source()** function in the RevBayes console:

```
source("RevBayes_scripts/basics.Rev")
```

Nevertheless, you will learn more if you type in the commands directly.

Let's start with the basic concepts for the interactive use of RevBayes with Rev (the language of RevBayes). You should try to execute the statements step by step, look at the output and try to understand what and why things are happening. We start with some simple concepts to get familiar and used to RevBayes. By now you should have executed RevBayes and you should see the command prompt waiting for input. The best exercise is to write these statements exactly in RevBayes.

Rev is an interpreted language for statistical computing and analyses in evolutionary biology. Therefore, the basics are simple mathematical operations, such as

```
# Simple mathematical operators:
1 + 1                          # Addition
10 - 5                         # Subtraction
5 * 5                          # Multiplication
10 / 2                         # Division
2^3                            # Exponentiation
5%2                            # Modulo
```

Just as a side note, you can also write multiple statements in the same line if you separate these by a semicolon (;). The statements will be executed as if you wrote each on a single line.

```
1 + 1; 2 + 2                  # Multiple statements in one line
```

Here you can see that comments always start with the hash symbol (#). Everything after the '#'-symbol will be ignored. In addition to these simple mathematical operations, we provide some standard math functions which can be called by:

```
# Math-Functions
exp(1)                          # exponential function
ln(1)                           # logarithmic function with natural base
sqrt(16)                        # square root function
power(2,2)                      # power function: power(a,b) = a^b
```

Notice that `Rev` is case-sensitive. That means, `Rev` distinguishes upper and lower case letter for both variable names and function names. For example, only the first of these two calls will work

```
exp(1)                          # correct lower case name
Exp(1)                          # wrong upper case name
```

Moreover, we provide functions for the common statistical distributions.

```
# distribution functions
dexp(x=1,lambda=1)     # exponential distribution density function
qexp(0.5,1)            # exponential distribution quantile function
rexp(n=10,1)           # random draws from an exponential distribution
dnorm(-2.0,0.0,1.0)    # normal distribution density function
rnorm(n=10,0,1)        # random draws from a normal distribution
```

You may have noticed that we sometimes provided labels of the arguments and sometimes not. You can always provide the argument labels and then `RevBayes` will match the arguments based on the labels.

```
dnorm(x=0.5,mean=0.0,sd=1)    # normal distribution density function
```

If you do not provide the argument labels, then `RevBayes` will match the arguments by the best fitting types and the order in which you provided the arguments.

```
dnorm(0.5,0.5,1)       # correct order
dnorm(0.5,1,0.5)       # mismatched order
```

You may provide also just some arguments with labels and leave the other arguments without labels.

```
dnorm(0.0,x=0.5,sd=1)  # partially labeled
```

If you do not remember what the parameter name or parameter names of a function are, then you can simply type in the function name and `RevBayes` will tell you the possible parameters with their names.

```
dnorm
```

2.2 Variable Declaration

The next, and very important feature of RevBayes, is variable declaration. We have three types of (model) variables, namely constant, deterministic and stochastic variables, which represent the same three types of DAG nodes. Here we show how to construct the different variables and how they behave differently. First, we focus on the difference between constant and deterministic variables.

Let us begin by creating a constant variable with name **a** and assigned the value 1 to it. The left arrow assignment (**<-**) always creates a constant variable.

```
# Variable assignment: constant and deterministic
a <- 1                          # assignment of constant node 'a'
```

You see the value of 'a' by just typing in the variable name and pressing enter.

```
a                               # printing the value of 'a'
```

If you want to see which type of variable (constant, deterministic or stochastic) 'a' has, then call the structure function for it.

```
str(a)                          # printing the structure information of 'a'
   _variable    = a
   _RevType     = Natural
   _RevTypeSpec = [ Natural, Integer, RevObject ]
   _value       = 1
   _dagType     = Constant DAG node
   _children    = [ ]
   .methods = void function ()
```

An additional quite useful built-in function in RevBayes is the **type** function which gives you only the type information of the variable and thus is a subset of the **str** function.

```
type(a)                         # printing the type information of 'a'
    Natural
```

Next, we create a deterministic variable **b** using the **:=** assignment computed by **exp(a)** and another deterministic variable **c** computed by **ln(b)**. Deterministic variables are always created using the colon-equal assignment (**:=**).

```
b := exp(a)                    # assignment of deterministic node 'b' with the
    exponential function with parameter 'a'
b                              # printing the value of 'b'
c := ln(b)                     # assignment of deterministic node 'c' with logarithmic
    function with parameter 'b'
c                              # printing the value of 'c'
```

Again, you see the type of the variable and additional information such as which the parents and children are by calling the structure function on it.

```
str(b)                         # printing the structure information of 'b'
```

For example, see the difference to the creation of variable 'd', which is a constant variable.

```
d <- ln(b)                     # assignment of constant node 'd' with the value if the
    logarithmic function with parameter 'b'
d                              # printing the value of 'd'
str(d)                         # printing the structure information of 'd'
```

Currently, the variables **c** and **d** have the same value. We can check this using the equal comparison (**==**).

```
e := (c == d)
e
```

Now, if we assign a new value to variable **a**, then naturally the value of **a** changes. This has the consequence that all deterministic variables that use 'a' as a parameter, i.e., the variable **b**, change their value automatically too.

```
a <- 2                         # reassignment of variable a; every deterministic node
    which has 'a' as a parameter changes its value
a                              # printing the value of 'a'
b                              # printing the value of 'b'
c                              # printing the value of 'c'
d                              # printing the value of 'd'
e
```

Since variable **d** was a constant variable it did not change its value. This also means that **e** is now false.

Finally, we show you how to create the third type of variables in Rev: the stochastic variables. We will create a random variable **x** from an exponential distribution with parameter **lambda**. Stochastic assignments use the **~** operation.

```
# Variable assignment: stochastic
lambda <- 1                    # assign constant node 'lambda' with value '1'
x ~ dnExponential(lambda)      # create stochastic node with exponential distribution
    and parameter 'lambda'
```

The value of **x** is a random draw from the distribution. You can see the value and the probability (or log-probability) of the current value under the current parameter values by

```
x                              # print value of stochastic node 'x'
x.probability()                # print the probability if 'x'
x.lnProbability()              # print the log-probability if 'x'
str(x)                         # printing all the information of 'x'
```

Similarly, we create a random variable **y** from a normal distribution by

```
mu <- 0
sigma <- 1
y ~ dnNorm(mu,sigma)
y.probability()                # print the probability of 'y'
y.lnProbability()              # print the log-probability if 'y'
str(y)                         # printing all the information of 'y'
```

Now you know everything there is about creating the different types of variables and the different ways in which these variables behave.

Simple variable manipulation and other types of assignments

Rev provides some convenience variable manipulation operations that are equivalent to variable manipulations in other programming languages such as C/C++, Java and Python. You can increment (**++**) and decrement (**-**) a variable. The increment operation increases the current value of a variable by 1 and the decrement operation decreases the value by 1. A post increment (**a++**) increases the value after returning the value, that is, the old value is returned. A pre increment (**++a**) increases the value before returning the value, that is, the new value is returned.

```
index <- 1
index++                        # post increment
++index                        # pre increment
index--                        # post decrement
--index                        # pre decrement
```

Additionally, you can use addition (`a += b`), subtraction (`a -= b`), multiplication (`a *= b`) and division (`a /= b`) to an existing variable.

```
index += 10                    # add 10 to the current value
index *= 2                     # double the current value
```

These variable manipulations will come in very handy for indices of vectors/arrays.

Vectors

Common values in `RevBayes` are of scalar types. That means, that not everything is a vector by default. Instead, you can create a vector using three different ways. First, you can call the vector function.

```
v <- v(1,2,3)                  # create a vector
```

Interestingly, we can use the same name for a variable as for a function: the variable `v` and the function `v(...)`. Both will still be fully functional and our interpreter checks if you asked for a function or a variable.

Second, you can use the square bracket notation.

```
w <- [1,2,3]                   # create a vector
```

And third, you can implicitly create the vector by assigning elements.

```
z[1] <-1                       # implicit creation of a vector
z[2] <-2
z[3] <-3
```

The implicit creation does not need to instantiate the variable beforehand. There are other useful built-in functions that produce vectors.

```
1:10                           # range function
rep(10,1)                      # replicate an element n times
seq(1,20,2)                    # built a sequence from a to b by c
```

Vectors in `Rev` belong to the class of objects that have methods. You can call a member method by

```
x.<method name>(<arguments>)
```

You have seen two methods previously, **probability** and **lnProbability**. If you don't remember what the methods were called, or if this object has any member methods, then you can get these by

```
v.methods()
```

In general, this is very, very useful. So for a vector we can get the size — the number of elements — by calling its member function:

```
v.size()
```

Control Structures

In this next part we will learn about control structures in **Rev**. The first control structure that we will look at is the **for** loop. **for** loop execute a single statement or a block of

```
# loops
for (<variable> in <set of value>) <single statement>

for (<variable> in <set of value>)
   <single statement>

for (<variable> in <set of value>) {
   <multiple statements>
   <multiple statements>
   <multiple statements>
}
```

The statement(s) will be execute for each value of variable of the **for** loop. A simple example is a **for** loop that computes the sum of

```
sum <- 0
for (i in 1:100) {
   sum <- sum + i
}
sum
```

Another example using a **for** loop is the computation of the Fibonacci number for a given integer.

```
# Fibonacci series using a for loop
fib[1] <- 1
fib[2] <- 1
for (j in 3:10) {
   fib[j] <- fib[j - 1] + fib[j - 2]
}
fib
```

We could also compute the Fibonacci numbers using a **while** loop. The **while** loop continues to execute the statement(s) until the condition is wrong.

```
# Fibonacci series using a while loop
fib[1] <- 1
fib[2] <- 1
j <- 3
while (j <= 10) {
   fib[j] <- fib[j - 1] + fib[j - 2]
   j++
}
fib
```

User Defined Functions

In Rev you can write your own functions as well. The syntax for writing function is:

```
function <return value type> <function name> (<list of arguments>) { <statements> }
```

As a simple example, let's write a function that computes the square of a number. We expect that the function takes in any real number. The type of real number is **Real**. Since the square is always a positive real number, we choose the return to be **RealPos**

```
# simple square function
function RealPos square ( Real x ) { x * x }
```

Now we can call our own function the same way as we call other already built-in function in RevBayes.

```
a <- square(5.0)
a
```

As an exercise, let's write a function that computes the factorial of a natural number.

```
# function for computing the factorial
function Natural fac(i) {
   if (i > 1) {
      return i * fac(i-1)
   } else {
      return 1
   }
}
b <- fac(6)
b
```

Here you see that within your own function you can call your function as well, which is commonly called recursive function calls.

Now let us write a recursive function for the sum of numbers which we computed before using a **for** loop.

```
# function for computing the sum
function Integer sum(Integer j) {
   if (j > 1) {
      return j + sum(j-1)
   } else {
      return 1
   }
}
c <- sum(100)
c
```

We can do the same for our favorite example, the Fibonacci series.

```
# function for computing the fibonacci series
function Integer fib(Integer k) {
   if (k > 1) {
      return fib(k-1) + fib(k-2)
   } else {
      return k
   }
}
d <- fib(6)
d
```

Now that should be enough to get you going with our first example analyses.

3 Exercise: Poisson Regression Model for Airline Fatalities

This exercise will demonstrate how to approximate the posterior distribution of some parameters using a simple Metropolis algorithm. The focus here lies in the Metropolis algorithm, Bayesian inference, and model specification—but not in the model or the data. After completing this computer exercise, you should be familiar with the basic Metropolis algorithm, analyzing output generated from a MCMC algorithm, and performing standard Bayesian inference.

Model and Data

We will use the data example from **?** (Table 2). A summary is given in table 2.

Table 2: Airline fatalities from 1976 to 1985. Reproduced from (**?**; Table 2.2 on p. 69).

Year	1976	1977	1978	1979	1980	1981	1982	1983	1984	1985
Fatalities	24	25	31	31	22	21	26	20	16	22

These data can be loaded into `RevBayes` by typing:

```
observed_fatalities <- v(24,25,31,31,22,21,26,20,16,22)
```

The model is a Poisson regression model with parameters α and β

$$y \sim \text{Poisson}(\exp(\alpha + \beta * x))$$

where y is the number of fatal accidents in year x. For simplicity, we choose uniform priors for α and β.

$$\begin{aligned} \alpha &\sim \text{Uniform}(-10, 10) \\ \beta &\sim \text{Uniform}(-10, 10) \end{aligned}$$

The probability density can be computed in `RevBayes` for a single year by

```
dpoisson(y[i],exp(alpha+beta*x[i]))
```

Problems

Metropolis Algorithm

The source file for this sub-exercise **`airline_fatalities_part1.Rev`**.

Let us construct a Metropolis algorithm that simulates from the posterior distribution $P(\alpha, \beta|y)$. For simplicity of the calculations you can "normalize" the years, e.g.

```
x <- 1976:1985 - mean(1976:1985)
```

A common proposal distribution for $\alpha' \sim P(\alpha[i-1])$ is the normal distribution with mean $\mu = \alpha[i-1]$ and standard deviation $\sigma = \delta_\alpha$:

$$\alpha' \sim \text{norm}(alpha[i-1], delta\_alpha) \tag{1}$$

```
alpha_prime <- rnorm(1,alpha[i-1],delta_alpha)
```

A similar distribution should be used for β'.

```
delta_alpha <- 1.0
delta_beta <- 1.0
```

After you looked at the output of the MCMC, play around to find appropriate values for δ_α and δ_β.

Now we need to set starting values for the MCMC algorithm. Usually, these are drawn from the prior distribution, but sometimes if the prior is very uninformative, then these parameter values result into a likelihood of 0.0 (or log-likelihood of -Inf).

```
alpha[1] <- -0.01   # you can also use runif(-1.0,1.0)
beta[1] <- -0.01    # you can also use runif(-1.0,1.0)
```

Next, create some output for our MCMC algorithm. The output will be written into a file that can be read into R or Tracer (**?**).

```
# create a file output
write("iteration","alpha","beta",file="airline_fatalities.log")
write(0,alpha[1],beta[1],file="airline_fatalities.log",append=TRUE)
```

Note that we need a first iteration with value 0 so that Tracer can load in this file.

Finally, we set up a **for** loop over each iteration of the MCMC.

```
for (i in 2:10000) {
```

Within the **for** loop we propose new parameters value.

```
   alpha_prime <- rnorm(1,alpha[i-1],delta_alpha)[1]
   beta_prime <- rnorm(1,beta[i-1],delta_beta)[1]
```

For the newly proposed parameter values we compute the prior ratio. In this case we know that the prior ratio is 0.0 as long as the new parameters are within the limits.

```
   ln_prior_ratio <- dunif(alpha_prime,-10.0,10.0,log=TRUE) + dunif(beta_prime
       ,-10.0,10.0,log=TRUE) - dunif(alpha[i-1],-10.0,10.0,log=TRUE) - dunif(beta[i
       -1],-10.0,10.0,log=TRUE)
```

Similarly, we compute the likelihood ratio for each observation.

```
   ln_likelihood_ratio <- 0
   for (j in 1:x.size() ) {
      lambda_prime <- exp( alpha_prime + beta_prime * x[j] )
      lambda <- exp( alpha[i-1] + beta[i-1] * x[j] )
      ln_likelihood_ratio += dpoisson(observed_fatalities[j],lambda_prime) - dpoisson(
          observed_fatalities[j],lambda)
   }
   ratio <- ln_prior_ratio + ln_likelihood_ratio
```

And finally we accept or reject the newly proposed parameter values with probability **ratio**.

```
   if ( ln(runif(1)[1]) < ratio) {
      alpha[i] <- alpha_prime
      beta[i] <- beta_prime
   } else {
      alpha[i] <- alpha[i-1]
      beta[i] <- beta[i-1]
   }
```

Then we log the current parameter values to the file by appending the file.

```
   # output to a log-file
   write(i-1,alpha[i],beta[i],file="airline_fatalities.log",append=TRUE)
}
```

As a quick summary you can compute the posterior mean of the parameters.

```
mean(alpha)
mean(beta)
```

You can also load the file into R or Tracer to analyze the output.

In this section of the first exercise we wrote our own little Metropolis algorithm in Rev. This becomes very cumbersome, difficult and slow if we'ld need to do this for every model. Here we wanted to show you only the basic principle of any MCMC algorithm. In the next section we will use the built-in MCMC algorithm of RevBayes.

MCMC analysis using the built-in algorithm in RevBayes

Before starting with this new approach it would be good if you either start a new RevBayes session or clear all previous variables using the **clear** function. Currently we may have some minor memory problems and if you get stuck it may help to restart RevBayes.

We start by loading in the data to RevBayes.

```
observed_fatalities <- v(24,25,31,31,22,21,26,20,16,22)
x <- 1976:1985 - mean(1976:1985)
```

Then we create the parameters with their prior distributions.

```
alpha ~ dnUnif(-10,10)
beta ~ dnUnif(-10,10)
```

It may be good to set some reasonable starting values especially if you choose is very uninformative prior distribution. If by chance you had starting values that gave a likelihood of -Inf, then RevBayes will try several times to propose new starting values drawn from the prior distribution.

```
# let us use reasonable starting value
alpha.setValue(0.0)
beta.setValue(0.0)
```

Our next step is to set up the moves. Moves are algorithms that propose new values and know how the reset the values if the proposals are rejected. We use the same sliding window move as we implemented above by ourselves.

```
mi <- 0
moves[mi++] = mvSlide(alpha)
moves[mi++] = mvSlide(beta)
```

Then we set op the model. This means we create a stochastic variable for each observation and clamp its value with the observed data.

```
for (i in 1:x.size() ) {
    lambda[i] := exp( alpha + beta * x[i] )
    y[i] ~ dnPoisson(lambda[i])
    y[i].clamp(observed_fatalities[i])
}
```

We can now create the model by pulling the up the model graph from any variable that is connected to our model graph.

```
mymodel = model( alpha )
```

We also need some monitors that report the current values during the MCMC run. We create two monitors, one printing all numeric non-constant variables to a file and one printing some information to the screen.

```
monitors[1] = mnModel(filename="output/airline_fatalities.log",printgen=10, separator
    = " ")
monitors[2] = mnScreen(printgen=10, alpha, beta)
```

Finally we create an MCMC object. The MCMC object takes in a model object, the vector of monitors and the vector of moves.

```
mymcmc = mcmc(mymodel, monitors, moves)
```

On the MCMC object we call its member method **run** to run the MCMC.

```
mymcmc.run(generations=3000)
```

And now we are done ☺

Posterior Distribution of α and β

Report the posterior mean and 95% credible intervals for α and β. Additionally, plot the posterior distribution of α and β by plotting a histogram of the samples. You can use the `R` function

Plot the curve of $m(x) = \mathrm{E}[\exp(\alpha + \beta * x)|y]$ for $x = [1976, 1985]$. You can generate draws from the posterior distribution of the expected value for a specific x by recording the current expected value at a

iteration i of the Metropolis algorithm $m\_sample(x)[i] = \mathrm{E}[\exp(\alpha[i] + \beta[i] * x)|y]$ and taking the mean of those samples (`m(x) = mean(m_sample(x))`) afterwards. Since `RevBayes` provides you with the samples of $m(x) = \mathrm{E}[\exp(\alpha + \beta * x)|y] = \lambda_x$ you can simply plot these posterior curves.

Produce a histogram of the predictive distribution of the number of fatalities in 2014 and estimate the posterior mean. The predictive distribution can be approximated simultaneously with the Metropolis algorithm. This means, for any iteration i you simulate draws from the conditional distribution for $x = 2014$ and the current values of $\alpha[i]$ and $\beta[i]$.

Estimate the distribution of the mean of the posterior predictive distribution of the the number of fatalities in 2014. Therefore, let us denote the expected value of the posterior distribution by μ. Since we do not know this value μ exactly, we can follow the Bayesian approach and associate a probability for each value m as being the true expected value of the posterior distribution, given the observations y ($P(m = \mu|y)$). You can be approximate this distribution by recording the expected value for the number of fatalities in 2014 ($\mathrm{E}[\exp(\alpha + \beta * x)|y]$) in each iteration i of the Metropolis algorithm. Plot a histogram of the expected values, compute the mean of the expected values and compare it to the previously obtained estimate of the mean of the posterior predictive distribution.

Follow the same approach as for the posterior predictive distribution for $x = 2015$, but this time for $x = 2016$ and estimate the probability of no fatality.

4 Exercise: Poisson Regression Model for Coal-mine Accidents

We will analyze a dataset coal-mine accidents. The values are the dates of major (more than 10 casualties) coal-mining disasters in the UK from 1851 to 1962.

A model for disasters

A common model for the number of events that occur over a period of time is a Poisson process, in which the number of events in disjoint time-intervals are independent and Poisson-distributed. We will discretize and look at the yearly number of accidents.

In order to take into account the possible change of rate, we will allow for different rates before and after year θ, where θ is unknown to us. Thus, the observation distribution of our model is $y_t \sim Poisson(\lambda_t)$ with $t = 1851, \ldots, 1962$ and

$$\lambda_t = \begin{cases} \beta & \text{if } t < \theta \\ \gamma & \text{if } t \geq \theta \end{cases}$$

Thus, the rate t is defined by three unknown parameters: β, γ and θ. A hierarchical choice of priors is given by

$$\begin{aligned} \eta &\sim Gamma(10.0; 20.0) \\ \beta &\sim Gamma(2.0; \eta) \\ \gamma &\sim Gamma(2.0; \eta) \\ \theta &\sim Uniform(1852, \ldots, 1962) \end{aligned}$$

which brings an additional parameter η in the model. For θ we have used an uniform prior over the years, but excluded year 1851 in order to make sure at least one year has rate β. The hierarchical prior carry the belief that β and γ are somewhat similar in size, since they both depend on η.

The model in Rev

We start as usual by loading in the data.

```
observed_fatalities <- v(4, 5, 4, 1, 0, 4, 3, 4, 0, 6, 3, 3, 4, 0, 2, 6, 3, 3, 5, 4,
   5, 3, 1, 4, 4, 1, 5, 5, 3, 4, 2, 5, 2, 2, 3, 4, 2, 1, 3, 2, 2, 1, 1, 1, 1, 3, 0, 0,
    1, 0, 1, 1, 0, 0, 3, 1, 0, 3, 2, 2, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 2, 1, 0, 0,
   0, 1, 1, 0, 2, 3, 3, 1, 1, 2, 1, 1, 1, 1, 2, 3, 3, 0, 0, 0, 1, 4, 0, 0, 0, 1, 0, 0,
    0, 0, 0, 1, 0, 0, 1, 0, 1)
year <- 1851:1962
```

In Rev we specify this prior choice by

```
eta ~ dnGamma(10.0,20.0)
beta ~ dnGamma(2.0,eta)
gamma ~ dnGamma(2.0,eta)
theta ~ dnUnif(1852.0,1962.0)
```

Then we select moves for each parameter. For the rate parameters — which are defined only on the positive real line — we choose a scaling move. Only for **theta** we choose the sliding window proposal.

```
mi <- 0
moves[mi++] = mvScale(eta)
moves[mi++] = mvScale(beta)
moves[mi++] = mvScale(gamma)
moves[mi++] = mvSlide(theta)
```

Then, we set-up the model by computing the conditional rate of the Poisson distribution, creating random variables for each observation and attaching (clamping) data to the variables.

```
for (i in 1:year.size() ) {
    rate[i] := ifelse(theta > year[i], beta, gamma)
    y[i] ~ dnPoisson(rate[i])
    y[i].clamp(observed_fatalities[i])
}
```

Finally, we create the model object from the variables, add some monitors and run the MCMC algorithm.

```
mymodel = model( theta )

monitors[1] = mnModel(filename="output/coal_accidents.log",printgen=10, separator = "
    ")
monitors[2] = mnScreen(printgen=10, eta, lambda, gamma, theta)

mymcmc = mcmc(mymodel, monitors, moves)

mymcmc.run(generations=3000)
```

Batch Mode

If you wish to run this exercise in batch mode, the files are provided for you.

You can carry out these batch commands by providing the file name when you execute the **rb** binary in your unix terminal (this will overwrite all of your existing run files).

- **$ rb RevBayes_scripts airline_fatalities_part1.Rev**
- **$ rb RevBayes_scripts airline_fatalities_part2.Rev**
- **$ rb RevBayes_scripts coalmine_accidents.Rev**

Useful Links

- RevBayes: https://github.com/revbayes/code

Questions about this tutorial can be directed to:

- Sebastian Höhna (email: sebastian.hoehna@gmail.com)
- Tracy Heath (email: tracyh@berkeley.edu)
- Michael Landis (email: mlandis@berkeley.edu)

References

Gilks, W., A. Thomas, and D. Spiegelhalter. 1994. A language and program for complex Bayesian modelling. The Statistician 43:169–177.

Höhna, S., T. A. Heath, B. Boussau, M. J. Landis, F. Ronquist, and J. P. Huelsenbeck. 2014. Probabilistic Graphical Model Representation in Phylogenetics. Systematic Biology 63:753–771.

Jordan, M. 2004. Graphical models. Statistical Science 19:140–155.

Koller, D. and N. Friedman. 2009. Probabilistic Graphical Models: Principles and Techniques. The MIT Press, Cambridge.

Lunn, D., D. Spiegelhalter, A. Thomas, and N. Best. 2009. The bugs project: Evolution, critique and future directions. Statistics in medicine 28:3049–3067.

Lunn, D. J., A. Thomas, N. Best, and D. Spiegelhalter. 2000. Winbugs-a bayesian modelling framework: concepts, structure, and extensibility. Statistics and computing 10:325–337.

Version dated: January 16, 2015