Chapter 1

Stochastic Models in Cartoons

Let's make some graphical animations to illustrate the BM and OU model. We have all the tools to do this in R, to help us understand the differences between the models and how we can exploit their features to explore evolutionary scenarios.

1.1 Brownian motion model

We can describe a Brownian motion process using the following equation:

$$dX(t) = \sigma\, dB(t). \tag{1.1}$$

This equation says that the change in value of X in some small interval in time is determined by a random amount of change $dB(t)$ times a scaling factor σ. That is, over a small increment of time, $dX(t)$ is the infinitesimal change in the character X over the infinitesimal interval from time t to time $t + dt$. The term $dB(t)$ is "white noise"; that is, the random variables $dB(t)$ are independent and identically-distributed normal random variables, each with mean zero and variance dt.

We can mimic this behavior by a simulation in discrete time. If we take the above equation and approximate it as:

$$\Delta X = X_{i+1} - X_i = \sigma dB(t) \tag{1.2}$$

$$X_{i+1} = X_i + \sigma dB(t) \tag{1.3}$$

We can see that a simulation is simply taking the old value of X_i and adding a random amount of change to it to get the next value (X_{i+1}), scaled (multiplied) by a constant value σ. If $\sigma = 1$, then it is in its most simplest form and we can code this simply as:

```
>    x[i+1] <- x[i] + devs[i]
```

Where `devs` is a random deviate. So let's do it! Start with 100 generations, and make 100 random draws.

```
> ngens = 100       # number of generations in our simulation
> devs =rnorm(ngens)    # 100 draws from a normal distribution or ``deviates"
```

Create a vector for our phenotype `x`, and fill it by starting from 0, and adding a random deviate to `x` at each generation.

```
> x <- c(0:100)    # initialize x (create a vector of the appropriate length), our phe
> for (i in 1:ngens)    # a loop which will run ngens times, each time increasing the
+ {
+    x[i+1] <- x[i] + devs[i]
+ }
```

```
> x      # take a look at the values of x, this is how x changes each generation
```

```
  [1]  0.000000000  0.744462138 -0.216999387  0.277856120 -0.850111862
  [6] -1.045048247  0.124522715  1.994246140  2.205044952  1.145236347
 [11]  0.722000790  0.111762633 -0.070591768  0.379538270 -0.540557529
 [16]  0.320167994 -0.628951227  0.897313139 -0.625029234  1.454085336
 [21]  1.485054936  2.494038803  2.557030189  3.119973868  4.153774817
 [26]  2.939491983  2.661497197  0.645652361  0.916461859  1.712422724
 [31]  1.567829823 -0.008616502 -1.055501233 -2.042286346 -1.921369177
 [36] -2.541656751 -1.527155194 -0.660870854 -2.345543414 -1.984317043
 [41] -1.210537259 -2.553489274 -2.864482182 -1.374771512  0.017544791
 [46]  0.400916346 -0.512301506 -0.396377305 -0.466476620 -1.387386283
 [51] -2.002001457 -1.803606005 -1.266239782 -2.124252317 -2.296738038
 [56] -0.528885592 -0.815100953  0.160312384 -1.330286139 -2.549568164
 [61] -2.313574420 -2.776887146 -2.851452203 -2.859195351 -2.681308278
 [66] -2.151420135 -2.513683598 -3.034923120 -2.524896260 -2.300414295
 [71] -3.657710236 -2.936822999 -2.198246288 -2.343157252 -5.014247378
 [76] -3.435313675 -4.315546247 -4.068479751 -6.193193538 -7.094018953
 [81] -7.500748544 -6.812502078 -6.460119097 -5.612358045 -7.337165206
 [86] -6.848191225 -6.885888016 -5.932932776 -5.642627458 -5.281215971
 [91] -6.643569466 -7.102560988 -8.076057099 -9.457997064 -8.242059483
 [96] -7.639423576 -7.206378679 -8.146783918 -8.004129281 -8.192078248
[101] -8.051097325
```

We can plot this single random walk. The function `plot()` creates a new plot window (and plots data, except that we told it to plot nothing by the parameter `type="n"` – we did this in order to plot the simulation slowly. The actually simulations are added one generation at a time by the command `lines()`.

```
> plot(1:length(devs), devs, type = "n", col="red",
+ ylim = c(-max(devs)*30, max(devs)*30),
+ xlab="Time", ylab="Value", main="BM Simulation")
> x <- c(0:100)
> for (i in 1:ngens)
+ {
+    x[i+1] <- x[i] + devs[i]
+    lines(i:(i+1), x[i:(i+1)], col="red")     # plotting the simulation
+ }
```

We can add σ in now simply by modifying the line of code specifying the BM:

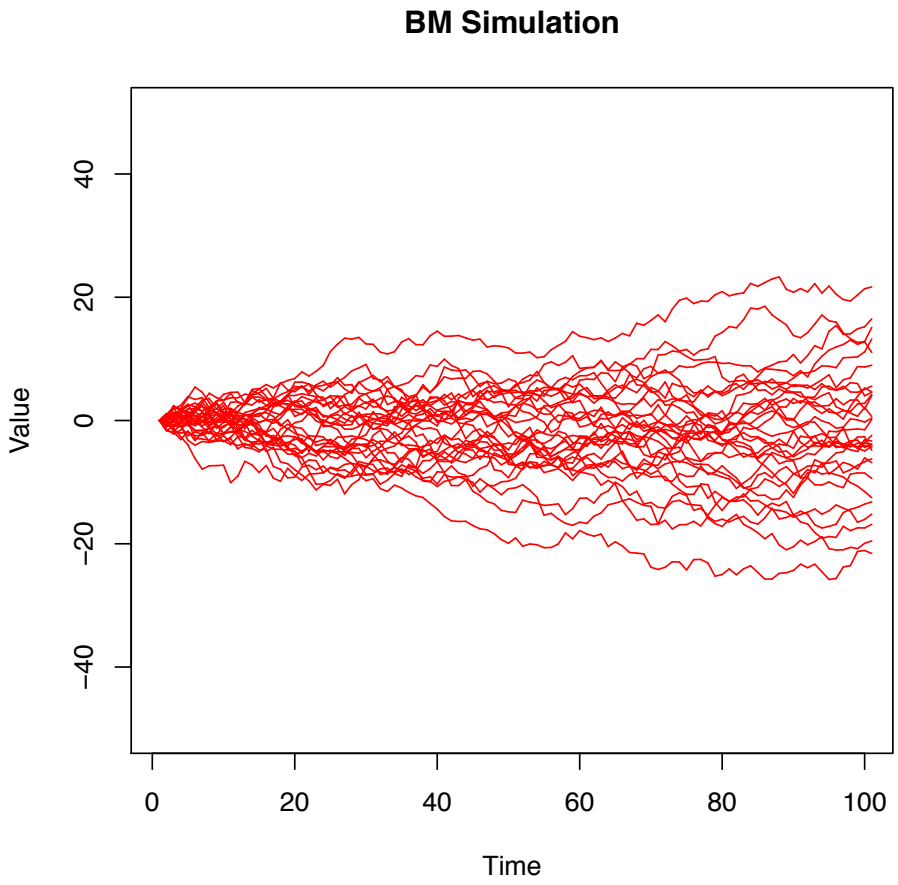
```
>    x[i+1] <- x[i] + sigma * devs[i]
```

In order to do 30 random walks (30 lineages), we need to place an outer loop, once for each random walk:

```
> bm.plot <- function( sigma=1, ngens=100, nwalks=30, yylim=c(-50, 50) )
+ {
+     plot(1:length(devs), devs, type = "n", col="red", ylim = yylim,
+      xlab="Time", ylab="Value", main="BM Simulation")
+
+     for (i in 1:nwalks)       # number of lineages to simulate
+     {
+         x <- c(0:ngens)
+         devs =rnorm(ngens)    # draws from a normal distribution, one per generation
+         for (i in 1:ngens)
+         {
+            x[i+1] <- x[i] + sigma*devs[i]     # BM equation
+                   # step through time, increasing x a little bit each time
+            lines(i:(i+1), x[i:(i+1)], col="red")   # plot line segment
+         }
+     }
+ }
```

Now run the simulation:

```
> bm.plot()
```



Now that we have created a function, we can play with it by varying the parameters to see what they mean. For example: ■= bm.plot(sigma=4) see the difference? bm.plot(sigma=1, ngens=300) @

For the following exercises, it will be helpful to collect the final values of each lineage. To do this, we need to (1) collect the final values, and (2) return them from the function. We can do this by adding in the lines for `xfinal` below:

```
> bm.plot <- function( sigma=1, ngens=100, nwalks=30, yylim=c(-30, 30) )
+ {
+     plot(1:length(devs), devs, type = "n", col="red", ylim = yylim,
+     xlab="Time", ylab="Value", main="BM Simulation")
+     xfinal <- c(1:nwalks)      ### initialize xfinal
+
+     for (j in 1:nwalks)
+     {
+         x <- c(0:ngens)
+         devs =rnorm(ngens)
```

```
+         for (i in 1:ngens)
+         {
+            x[i+1] <- x[i] + sigma*devs[i]     # BM equation
+                   # step through time, increasing x a little bit each time
+            lines(i:(i+1), x[i:(i+1)], col="red")   # plot line segment
+         }
+         xfinal[j] <- x[ngens+1]       ### collect the last value at each lineage
+     }
+     return(xfinal)          ### return the final values
+ }
```

1.2 Exercises

The values at the end of the simulation represent the phenotypic value after evolving for *codengens* generations. Prove to yourself the following properties of Brownian Motion processes:

1. The mean value at the end of the process is centered on the value at the start of the process (although it may be very noisy from simulation to simulation, meaning you may have to do a lot of simulations to get a good estimate of the mean).
2. The variance grows in proportion to time. Remember that in the BM, the variance is given by $\sigma^2 t$.

1.3 R tricks – feel free to skip

The loop is easier to understand in terms of a stochastic process, but actually we can write this code much more compactly:

Adding up a series of BM steps using the cumulative sum function:

```
>     sigma=1
>     cumsum(rnorm(ngens, sd=sigma))
```

Plotting all the line segments at once:

```
>     y <- c(0, cumsum(rnorm(ngens, sd=sigma)) )
>     lines(0:ngens, y)
```

Or even more compactly:

```
>    sigma=1
>    lines(0:ngens, c(0, cumsum(rnorm(ngens, sd=sigma))))
```

And doing all of the lineages using lapply:

```
> bm.plot <- function( sigma=1, ngens=100, nwalks=30, yylim=c(-50, 50))
+ {
+ # Set up plotting environment
+         plot(0, 0, type = "n", xlab = "Time", ylab = "Trait",
+                 xlim=c(0, ngens), ylim=yylim)
+
+ # Draw random deviates and plot
+         lapply( 1:nwalks, function(x)
+           lines(0:ngens, c(0, cumsum(rnorm(ngens, sd=sigma)))))
+ }
```

If you want to show the simulations to screen, then you may actually prefer to do the slower for-loops, as the lapply is too fast.

1.4 The OU Process

Recall that we described a Brownian motion process using the following equation:

$$dX(t) = \sigma\, dB(t). \tag{1.4}$$

The simplest stochastic model that incorporates selection is the Ornstein-Uhlenbeck process Uhlenbeck and Ornstein (1930):

$$dX(t) = \alpha\,(\theta - X(t))\,dt + \sigma\, dB(t). \tag{1.5}$$

Eq. 2.2 adds two new parameters to our changing phenotype. The parameter α measures the strength of selection. When $\alpha = 0$, the deterministic part of the OU model drops out and (2.2) collapses to the familiar BM model of pure drift.

The OU process was first described in 1930. Russ Lande showed that the evolution of a species' mean phenotype could be described by an OU process, representing stabilizing selection Lande (1976), and was discussed by Felsenstein (1988). Thomas Hansen first recognized its usefulness in phylogenetic comparative analysis Hansen (1997) to describe adaptive evolution along a phylogeny. Butler and King developed this application further Butler and King (2004) and expanded upon the use of the OU model to represent alternative models of adaptive and non-adaptive evolution, as well as a rigorous model-comparison framework.

We can again write a simple simulation by copying the form of the equation, with parameters for α and θ:

```
>         x[i+1] = alpha*(theta-x[i])+x[i] + sigma*devs[i]
```

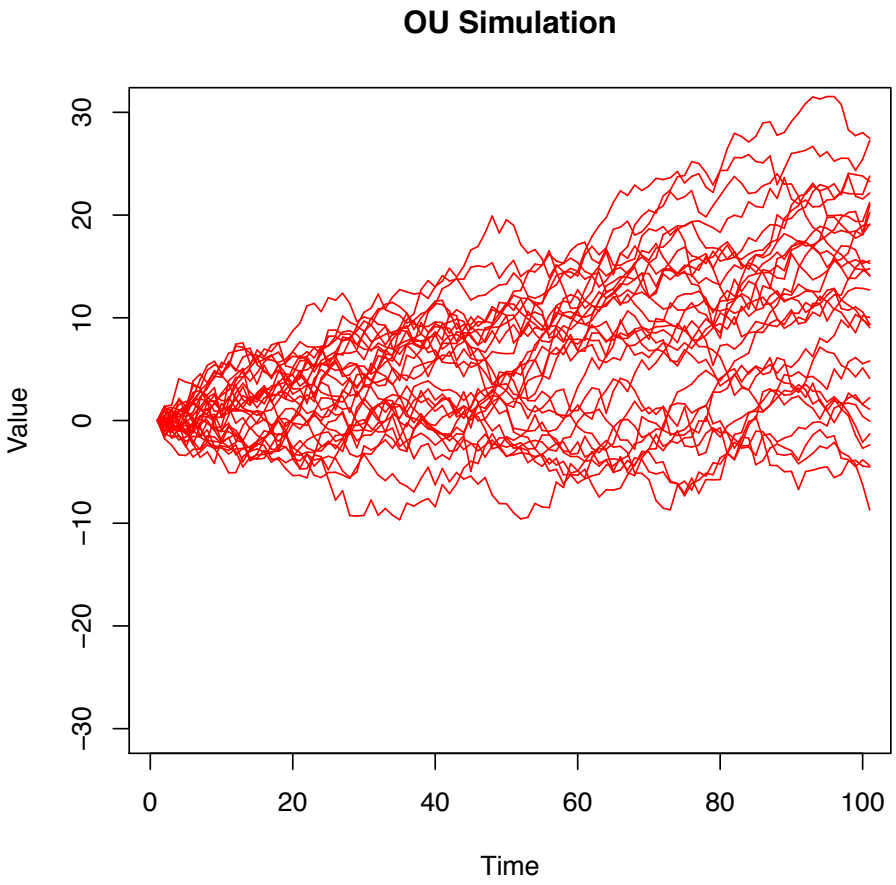
OK! Let's simulate the OU process:

```
> ou.plot <- function( alpha=0.005, theta=0, sigma=1, ngens=100, nwalks=30,
+ yylim=c(-30, 30) )
+ {
+     plot(1:length(devs), devs, type = "n", col="red",  ylim = yylim,
+     xlab="Time", ylab="Value", main="OU Simulation")
+     xfinal <- c(1:nwalks)
+
+     for (j in 1:nwalks)       # number of lineages to simulate
+     {
+         x <- c(0:ngens)
+         devs =rnorm(ngens)
+         for (i in 1:ngens)
+         {
+             x[(i+1)] = alpha*(theta-x[i])+x[i] + sigma*devs[i]    # OU eq
+             lines(i:(i+1), x[i:(i+1)], col="red")   # plot line segment
+         }
+         xfinal[j] <- x[ngens+1]   ## collect last value at each lineage
+     }
+     return(xfinal)          ### return the final values
+ }
```

You can run the function by typing:

```
> ou.plot( theta=30 )
```

```
 [1]  6.08921384  4.72895357 14.29856946  7.30704195 31.01381330  8.94375197
 [7] 19.67031117 14.26942261  8.42887230  5.01125743 29.38009569  3.43223404
[13]  6.12642411 21.23142466 25.83394865 15.02515763 18.18470002  2.36389169
[19]  0.29109179 15.57007748 10.65020548 10.72746658 11.46775078  8.33916492
[25] 10.38746828 18.72944767 15.54081632  0.06960877 12.65612399 13.64733399
```



Do you see any difference? The σ value is 1 as in the BM simulation, but here the default value of $\alpha = -.005$, and we set the optimum value as $\theta = 30$.

1.5 Exercises

1. What is the effect of bigger or smaller α? Does the simulation always reach the optimum value you set? What else could you change?

2. What is the effect of changes in θ?

3. What is the difference between BM simulations and OU simulations with similar σ? Try your own or do a BM simulation with $\sigma = 1$, and OU with $\sigma = 1$, $\alpha = 0.01$, and $\theta = 0$.

1.6 Using the OU to model adaptive evolution

Simpson (1953) painted a vivid picture of adaptive evolution in macroevolutionary time. He imagined lineages evolving through time within an adaptive zone. On rare occasions, the lineage may enter a new adaptive zone, which would result in morphological adaptations that reflect their new functional demands. In some cases, entering a new adaptive zone may even result in special bursts of evolution, perhaps adaptive radiation. However, we are getting ahead of ourselves.

We can use the OU to model adaptive evolution. Not only does it have a term for selection, but also for adaptive optima. These adaptive optima can be used to represent selective regimes, or lifestyles, or different suites of ecological pressures. While we limit each branch of the phylogeny to having only one optima, we can assign different optima to different branches, according to our biological hypothesis. For example, we may have a lizard lineage that is ancestrally insectivorous, but they may have evolved herbivory along some of its branches (Fig. 1.1).

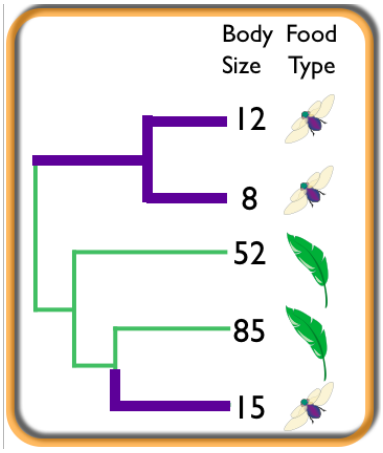


Figure 1.1: Hypothetical adaptive regimes. We group branches leading to herbivory in a separate adaptive regime relative to those leading to insectivory.

Note that we don't assign a unique optimum to every branch, that would result in too many parameters and not enough data to estimate them unambiguously. Rather, we are postulating a limited number of adaptive regimes – the simplest model that can explain most of the data.

Download the function `OU.sim.branch` and load it into your R workspace by the following command (make sure you download it to your working directory):

```
> source("OU.sim.branch.R")
```

If you'd like to see the function, just type it's name:

```
> OU.sim.branch
```

and the function will dump to screen. In particular look at the default parameters in the function call and do the exercises below.

1.7 Exercises

Is the OU process, as simple as it is, flexible enough to suit our needs? Let's try some simulations, this time with a branch along the phylogeny, and see if we can simulate changes in the phenotype.

1. Can you make the phenotype diverge in different branches of the phylogeny?
2. Food for thought. Does the simulation always reach the optimum? What are the possible explanations?

1.8 Making movies

In order to make a movie of the plot, you will need to save a series of plots as separate graphics files, similar to the "flip-books" you played with as a child. You need to make a plot with the first lineage, then the first two lineages, then the first three lineages, and so on.

So it would make sense to make a matrix to store the lineages, then plot through cumulatively:

```
> ngens=100
> nwalks=30
> sigma=1
> sims <- sapply(1:nwalks, function(x) c(0, cumsum(rnorm(ngens, sd=sigma))))
> yylim <- c(-30, 30)
> png(filename="movies/Rplot%03d.png")
>               # turn on png graphical device (write to file)
> for (i in 2:nwalks)
+ {
+         plot(0, 0, type = "n", xlab = "Time", ylab = "Trait",
+         xlim=c(0, ngens), ylim=yylim)
+         apply( sims[,1:i], 2 , function(x) lines(0:ngens, x, col="red"))
+ }
> dev.off()      # turn off png
```

Then in a terminal, move into the `movies` directory. If you have imagemagik installed:

convert -delay 10 Rplot.png Rplot.gif

To make a .mov file, you can use Quicktime Pro (but you have to pay for the Pro upgrade). In R version 2.8 there is a new package named `animation` which calls ImageMagick from R.

1.9 RGL graphics

The 3D animations that I showed were produced using the package `rgl`.

You can see the graph gallery at http://rgl.neoscientists.org/gallery.shtml. I have also included my source code in the wiki. Under `ou2drgl.R`.