Random Simulation

Simulation is way to learn the consequences of models that cannot be solved analytically. Very few models can be solved analytically, so simulation is part of most modeling and analysis, in all fields. If the model involves random variables, then the simulation must generate and use computer generated random numbers. The result of one random simulation is not the same as the next. It takes many simulations to build a picture of the model. The output of a random simulation is one or more random numbers. It takes many simulations to build a picture of the probability distribution of the output.

Random number generator

A *pseudo-random number generator* is an algorithm that produces numbers that look random. The numbers are not actually random. You get the same sequence every time you run the algorithm, if you use the same *seed*. The (pseudo)-random number generator has a *state*, `s`, which is an array of integers. To get the next "random" number in the sequence, the random number generator *updates* the state, producing a new state that depends on the old state in some complicated way that simulates randomness (without being random). The next (pseudo)-random number is some function of the state.

```
s = [some algorithm](s)  # update the state
x = [some formula](s)    # produce a (pseudo)-random number
```

If you don't do anything, the starting state comes from some information that is likely to be roughly random – typically the current second or millisecond from the computer's internal clock. The `R` function `set.seed(number)` generates a state, `s` from the given `number`. This makes it possible to repeat a pseudo-random number sequence. This is particularly helpful for debugging (finding and fixing mistakes in a script). If your algorithm does something wrong in some case, you want to be able to repeat that case to figure out what is going wrong.

`R` has a number of random number generators, for different probability distributions and situations. The function `rnorm(n)` produces n independent random variables with the *standard normal* density, which is Gaussian with mean zero and variance 1. The density is

$$p(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} \ .$$

The numbers are not actually random (as was said above), they're not independent and they don't exactly have the normal distribution $p(x)$. But you need a PhD in random number generators to know how to detect the difference (if you

use the *Mersenne twister*, which R always uses unless you tell it not to). Here is the random number generator in action, at the command line in the R console:

```
> rnorm(4)
[1] -0.8172679  0.7720908 -0.1656119  0.9728744
> rnorm(4)
[1] 1.7165340 0.2552370 0.3665811 1.1807892
> set.seed(17)
> rnorm(4)
[1] -1.01500872 -0.07963674 -0.23298702 -0.81726793
> rnorm(4)
[1]  0.7720908 -0.1656119  0.9728744  1.7165340
> set.seed(17)
> rnorm(4)
[1] -1.01500872 -0.07963674 -0.23298702 -0.81726793
> rnorm(4)
[1]  0.7720908 -0.1656119  0.9728744  1.7165340
```

First we ask for a sequence of length 4. The numbers look OK (about as many positive as negative, on the order of 1). Next we ask for another sequence. The second 4 look good too. Now we use `set.seed()` to specify the state of the random number generator. The next 4 standard normals look good, as does the sequence after it. Then we reset the state using the same `seed`, 17. This makes the random number generator produce the same sequence.

There are many distributions built into R. The *uniform* distribution is used in many applications. The probability density is

$$p(x) = \begin{cases} 1 & \text{if } 0 \le x \le 1 \\ 0 & \text{if } x < 0 \text{ or } x > 1 \, . \end{cases}$$

A uniform random variable is equally likely to be anywhere in the interval $[0, 1]$ and is never outside that interval. The R command `runif(n)` returns n independent uniform numbers. Figure 1 shows what happened when I asked R for 30 independent uniformly distributed numbers, at the command prompt in the console:

```
[19] 0.56331137 0.24699811
> runif(30)
 [1] 0.73432757 0.70365677 0.73682406 0.80186265 0.75207300 0.63377903 0.24760893 0.55133933 0.23476108
[10] 0.25861469 0.95288810 0.85687457 0.27566924 0.55452418 0.87694381 0.11880538 0.54673734 0.23275320
[19] 0.60226778 0.71253911 0.64890360 0.18756242 0.23839289 0.07038709 0.11948412 0.61811947 0.16848571
[28] 0.14597753 0.18971358 0.57549452
>
```

Figure 1: Asking R for 30 uniformly distributed random variables.

The numbers are all between 0 and 1, and seem like they might be uniformly distributed. The figure also explains what the `[1]` means when R returns a value at the console window. It's the first element of a list. The second line of the output starts with element 10 (count them). The third line starts with element 19.

Uniform random variables (supplied by `runif()` are used in simulating random processes that involve decisions. For example, in the binomial tree process $S \to uS$ with probability p_u and $S \to dS$ with probability p_d. If U is a random variable uniformly distributed in $[0, 1]$, then

$$\Pr(U \leq p_u) = p_u \ . \tag{1}$$

A *Bernoulli* random variable has only two possible values (`TRUE` or `FALSE`, 0 or 1, uS or dS), determined by a probability p. Here is an implementation in R. The random variable B will have the value `TRUE` with probability p and the value `FALSE` with the complementary probability $1 - p$. Note that `runif(n` returns an array of length n. `R` treats an array of length 1 as a number. That's why you can test `U <= p` instead of the more complicated but more correct `U[1] <= p`. If you make `U = runif(2)` and then test `( U <= p )`, you will get a warning. `R` (like most languages) has an `if, else` command:

```
if (...) {
    [ some commands ]
  } else {
    [ other commands ]
  }
```

If the condition `(...)` is `TRUE`, then it executes the commands in the first code block (`[ some commands ]`). If the condition is `FALSE`, then it executes the other code block (`[ other commands ]`).

```
U = runif(1)
if ( U <= p ) {
    X = 1
  } else{
    X = 0
  }
```

This will make $X = 1$ with probability p, and $X = 0$ with probability $1 - p$. You can do this with the `R` function `rbinom` (for *binomial*), but this form will be more convenient.

Simulating a random process

A *random process* (also called *stochastic* process) is a sequence of steps (a process) that are random. The binomial tree process is an example. Let S_k be the stock price after k time periods. The starting price S_0 is given but the prices S_1, S_2, ... are random. The random "evolution" of the process is given by the binomial tree model

$$S_{k+1} = \begin{cases} uS_k & \text{with probability } p_u \\ dS_k & \text{with probability } p_d = 1 - p_u \ . \end{cases} \tag{2}$$

The sequence of prices S_0, S_1, ..., S_n is a random *path*. One *simulation* of the process (2) creates one random path. Many simulations create many different random paths.

Figure 2 is an `R` script that makes a random binomial tree path. Lines 3 to 6 give the parameters of the binomial tree. Line 7 sets the starting price. Line 10 is part of an old programmer's trick/habit. At each step `k` (starting at line 11), there will be an old price, which is S_{k-1}. The new price created will be S_k. The first trip through the loop, we will have $k = 1$, so $S_{\text{old}} = S_{k-1} = S_0$. Line 10 sets that up. Lines 12 through 17 take one binomial tree step, using the logic given above. Line 18 records this step of the path. Line 19 gets ready for the next trip through the loop. The price that is new in this trip will be old in the next trip. Lines 22 and 23 create a formatted output line.

randDemo.R

```
1  #   Simulate a random binomial tree path
2
3  n = 10
4  u = 1.1
5  d = .9
6  p_u = .6
7  S0 = 100    # starting price
8  S  = 1:n    # allocate an array for the price path
9
10 S_Old = S0  # S_Old is the price at the previous time, initialized to S0
11 for ( k in 1:n){
12    U = runif(1)    # note, u and U are different
13    if ( U < p_u ) {
14       S_New = u*S_Old        # S -> uS with probability p_u
15     } else {
16      S_New = d*S_Old         # S -> dS with probability p_d = 1-p_u
17     }
18    S[k] = S_New              # record the next step in the path
19    S_Old = S_New             # what was new will be old in the next ...
20                              # ... trip through the loop
21 }
22 output = sprintf("Final price is %8.2f",S_Old)
23 cat(output)
```

Figure 2: A script to make a random binomial tree path.

Figure 3 shows what happens when you run this script a few times. The result is random, it can be different every time. There are finitely many possible outcomes in a binomial tree of length n (is it n or $n+1$?), so you expect repeats eventually. It may be surprising that $S_n = 116.23$ is repeated three times in just 9 trials, and two other values are repeated twice. Using probability theory (see Assignment 5), you can see that some outcomes are more likely than others.

```
> source("randDemo.R")
Final price is    95.10
> source("randDemo.R")
Final price is    77.81
> source("randDemo.R")
Final price is   116.23
> source("randDemo.R")
Final price is   116.23
> source("randDemo.R")
Final price is   116.23
> source("randDemo.R")
Final price is   142.06
> source("randDemo.R")
Final price is   173.63
> source("randDemo.R")
Final price is   142.06
> source("randDemo.R")
Final price is    95.10
>
```

Figure 3: A script to make a random binomial tree path.

Figure 4 is an R function using the code from Figure 2 that generates and returns a path. The function is defined on lines 5 to 9. It is a nice coding style (I feel) to put each argument on its own line with its own comment. Lines 11 to 15 calculate the parameters of the binomial tree from the parameters of the problem. The formulas are from class.

binPath.R

```
1  #   Function to simulate a binomial tree path
2  #   Return an array of length n+1 that includes S0 and...
3  #   ... n random path steps
4
5  binPath = function( n,        #  the number of random steps
6                      mu,       #  the expected return, in percent/year
7                      sig,      #  the vol, in percent/year
8                      S0,       #  the starting price
9                      T) {       #  simulate up to this time
10
11     dt = T/n
12     u  = 1 + sig*sqrt(dt)
13     d  = 1 - sig*sqrt(dt)
14     pu = ( 1 + ( mu*sqrt(dt)/sig) ) /2  # formula from class
15     pd = ( 1 - ( mu*sqrt(dt)/sig) ) /2
16
17     S_path    = 1:(n+1)          # initialize a path of length n+1
18     S_path[1] = S0               # the first value is the starting value
19                                  # make the path
20     S_Old     = S0
21     for ( k in 1:n){
22         U = runif(1)    # note, u and U are different
23         if ( U < p_u ) {
24             S_New = u*S_Old        # S -> uS with probability p_u
25           } else {
26             S_New = d*S_Old        # S -> dS with probability p_d = 1-p_u
27           }
28          S_path[k+1] = S_New       # record the next step in the path
29          S_Old = S_New             # what was new will be old in the next ...
30                                    # ... trip through the loop
31       }
32      return( S_path )
33 }
34
```

Figure 4: A script to make a random binomial tree path.

Figure 5 shows how to play with this function from the command line. The parameters are typical. The unformatted output is hard to read – don't do it this way. Each of the five paths starts with 100.0000 (Formatting would have allowed you to print fewer pointless zeros.). In the first step, four paths go up to 109.4.. and one goes down to 90.5... The path that went down to 90.5.. then goes up to 99.1... The first path goes up to 109.4.. then down to the same value, 99.1... This shows that the tree is recombining: $S_0 \cdot d \cdot u$ (the third path) is equal to $S_0 \cdot u \cdot d$ (the first path). The first and fourth paths end in the same place, $S_n = 115.6..$, but take different "paths" to get there. Both paths end at $S_0 u^6 d^d$, going up six times and down four times.

first path:	up,	down,	up,	down,	down,	up,	up,	down,	up,	up,
fourth path:	up,	up,	up,	down,	down,	down,	up,	up,	down,	up,

```
> source("binPath.R")
> n = 10
> mu = .1
> sig = .3
> T = 1
> S0 = 100
> binPath(n, mu, sig, S0, T)
 [1] 100.00000 109.48683  99.10000 108.50145  98.20810  88.89126  97.32423 106.55721  96.44831 105.59820
[11] 115.61612
> binPath(n, mu, sig, S0, T)
 [1] 100.0000 109.4868 119.8737 108.5015 118.7948 107.5249 117.7256 128.8941 116.6661 105.5982 115.6161
> binPath(n, mu, sig, S0, T)
 [1] 100.00000  90.51317  99.10000 108.50145 118.79480 130.06467 142.40369 155.91329 170.70452 154.51007
[11] 169.16818
> binPath(n, mu, sig, S0, T)
 [1] 100.00000 109.48683 119.87367 131.24588 118.79480 107.52494  97.32423 106.55721 116.66612 105.59820
[11] 115.61612
> binPath(n, mu, sig, S0, T)
 [1] 100.00000 109.48683 119.87367 131.24588 118.79480 107.52494  97.32423  88.09124  79.73417  87.29842
[11]  79.01656
>
```

Figure 5: A script to make a random binomial tree path.

Figure 6 plots five random paths (*realizations*) of the binomial tree process. The parameters are all in the plot. These were made with the script `PlotPaths.R`, which uses the function file `binPath.R`. These are posted with this handout. You can see (more easily in the plot than just with the numbers) that the tree is recombining. You can see that multiplying by u (or d) many times in a row produces a curved line, not a straight one. This is because it's part of an exponential, not a linear function. Note that the stock price axis (the vertical axis) goes from 50 on the low side to 200 on the high side. If $\mu = 0$, it should (we think, maybe??) be about as likely to multiply by 2 as to divide by 2.

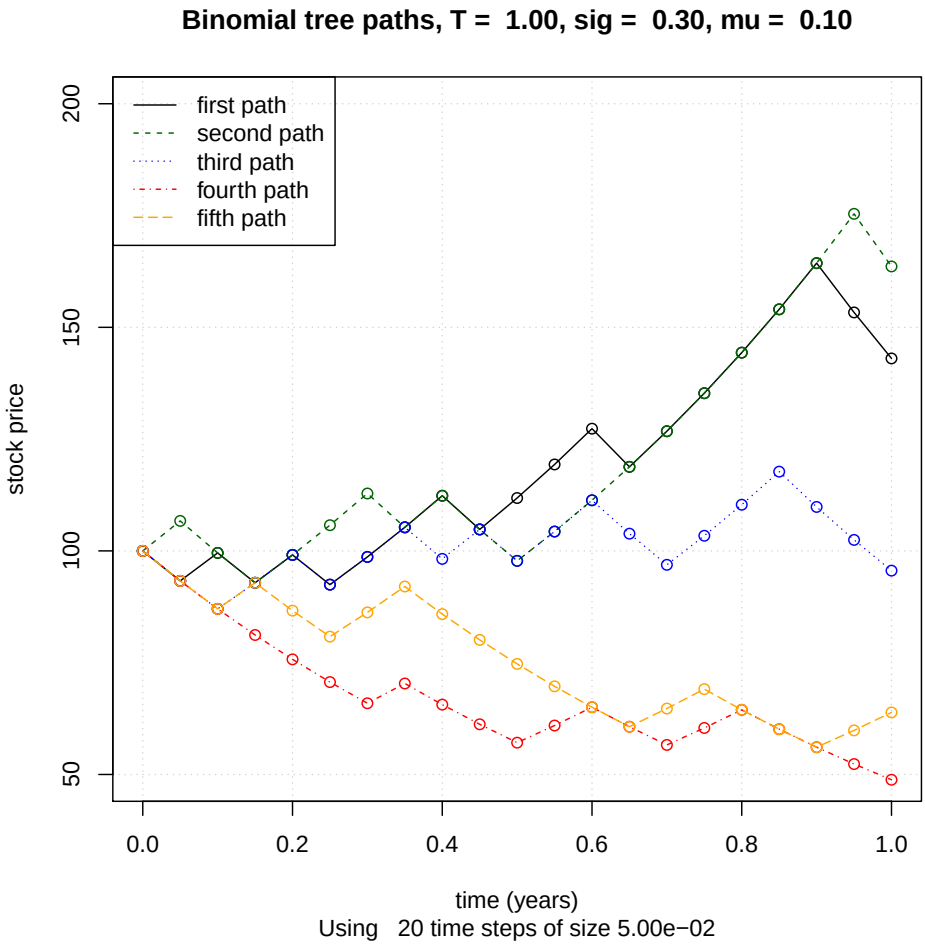

Figure 6: Binomial tree paths with $n = 20$ steps.

Figure 7 gives five paths with $n = 500$ steps in the same time interval. This means that $\Delta t = T/n$ is much smaller. The up and down multipliers u and d are much closer to 1 and the up and down probabilities p_u and p_d are closer to $\frac{1}{2}$. Thanks to lines 11 to 16 in the code of Figure 4, you can make this plot just changing n to 500 in one place in the code.

Figure 8 has Δt ten times smaller than Figure 7. You cannot see the individual circles on a trajectory, but they're still there in the code. The circles blend together to form thick random curves. The lowest end is the trajectory that takes $S_0 = 100$ to $S_T \sim 75$. On the high end are two trajectories that take $S_0 = 100$ to $S_T \sim 150$. If you run it again, you get five different curves, but they are in a similar range.

Some comments about the script `PlotPaths.R`. It has a lot of lines but it wasn't hard to create. I started with the plot script from the earlier handout `plotting.R`. That had a title with parameters (I changed the parameters, but it's still a `sprintf()` statement. It had a call to the plot function `plot()` to create one curve, the axes, title, etc. It had a call to `lines()`. I modified the call to `lines()` to print the second path. Then I copy/pasted that call three times to print paths 3, 4, and 5. I did some other copy/pasting to make the five calls to `binPath()` and other things. You don't measure the difficulty in creating a piece of software by the number of lines. It depends on how you create it and what code you start with.

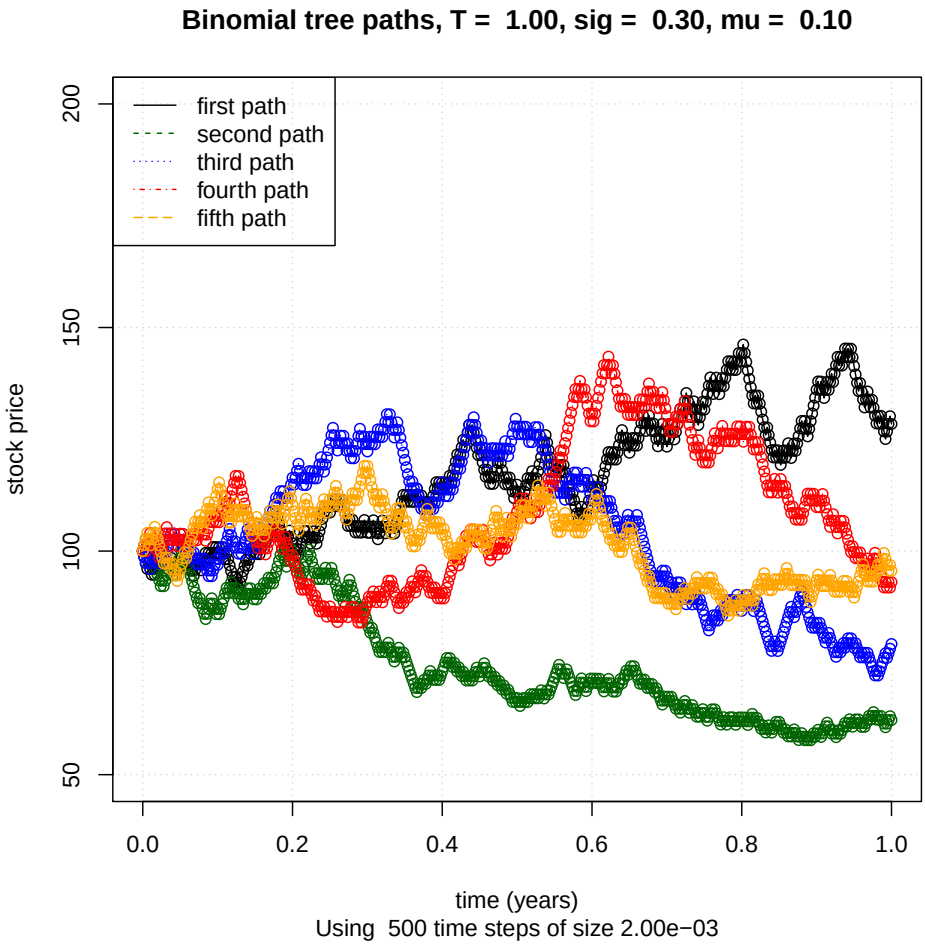

Figure 7: Binomial tree paths with $n = 500$ steps.

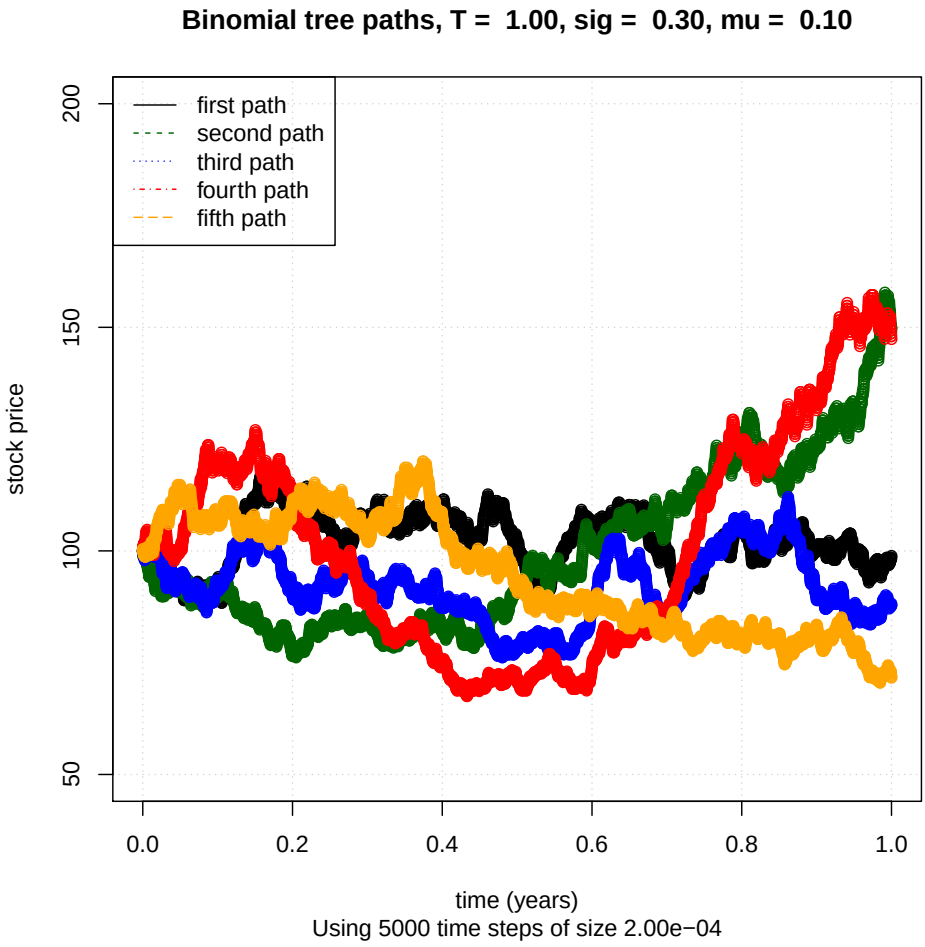

Figure 8: Binomial tree paths with $n = 5000$ steps.

Histogram of simulation data

Just five paths do not give a clear picture of the distribution of paths in the random binomial tree model. A *histogram* is a plot that reveals the distribution of random samples from a data set or from a sequence of simulations. Suppose you have m data values $X_1, \ldots, X_m$ and you want to learn their distribution. It is done by dividing the $x-$axis into *bins*. A bin is a small interval $B_k = [x_{k-1}, x_k]$. The *breakpoints* x_k are usually (but not always) separated by uniform spacing Δx, so $x_k = x_0 + k\Delta x$. The bins are $B_1 = [x_0, x_1]$, $B_2 = [x_1, x_2]$, etc.

The bin *counts* are the number of data values in each bin:

$$N_k = \#\{X_j \in B_k\} \ .$$

A graph of N_k as a function of k is a *histogram*.

A histogram reveals (with some noise and inaccuracy) the probability density $p(x)$ that the data samples X_j come from. If X is a random variable with probability density $p(x)$, then the probability for X to be in bin B_k is (exactly and then approximately)

$$p_k = \Pr(X \in B_k) = \int_{x_{k-1}}^{x_k} p(x)dx \approx \Delta x \ p(x_k) \ .$$

The probability to be between x_{k-1} and x_k is the area under $p(x)$ in that interval, which is approximately the width $\times$ the height, which is $\Delta x \cdot p(x_k)$. If you have m data samples from the density $p(x)$, then the expected number to land in B_k is

$$\mathrm{E}[N_k] = mp_k \ .$$

For large m, the law of large numbers says that mp_k is a reasonable approximation of $\mathrm{E}[N_k]$. If we use the approximate expression for p_k, this leads to the approximation

$$p(x_k) \approx \frac{N_k}{m\Delta x} \ . \tag{3}$$

In the graph, the difference between plotting N_k or plotting $N_k/(m\Delta x)$ is just the label on the vertical axis. Figures 9 and 10 have the vertical axis labelled as a probability density, which means that it's plotting the right side of (3).

The difference between the histograms in Figures 9 and 10 is only that the number of paths in Figure 10 is 100 times more. For some reason, the `R` histogram function `hist()` chose bigger bins for Figure 10 even though I asked for the same number of bins. Comparing the figures, you can see that using 100 times more paths makes the histogram less noisy. But it still is noisy. It isn't easy to estimate probability densities.

Figure 11 shows the script that made these histograms. Lines 4 to 8 are parameters, as before. Line 9 sets the number of paths. Lines 11 to 16 generate m sample paths and record the final price for each path in an array called `S`. After that it's graphics as before. Line 24 says to use 30 bins. Line 25 says to label the vertical axis as probability density (using formula (3) or something close to it – the documentation isn't clear on this point).

It took quite a while for my three year old laptop to run the program. I could have coded it to run faster, but this isn't a coding class. It would be even faster in a compiled programming language like C++. But still, simulations like this tend to be slow.

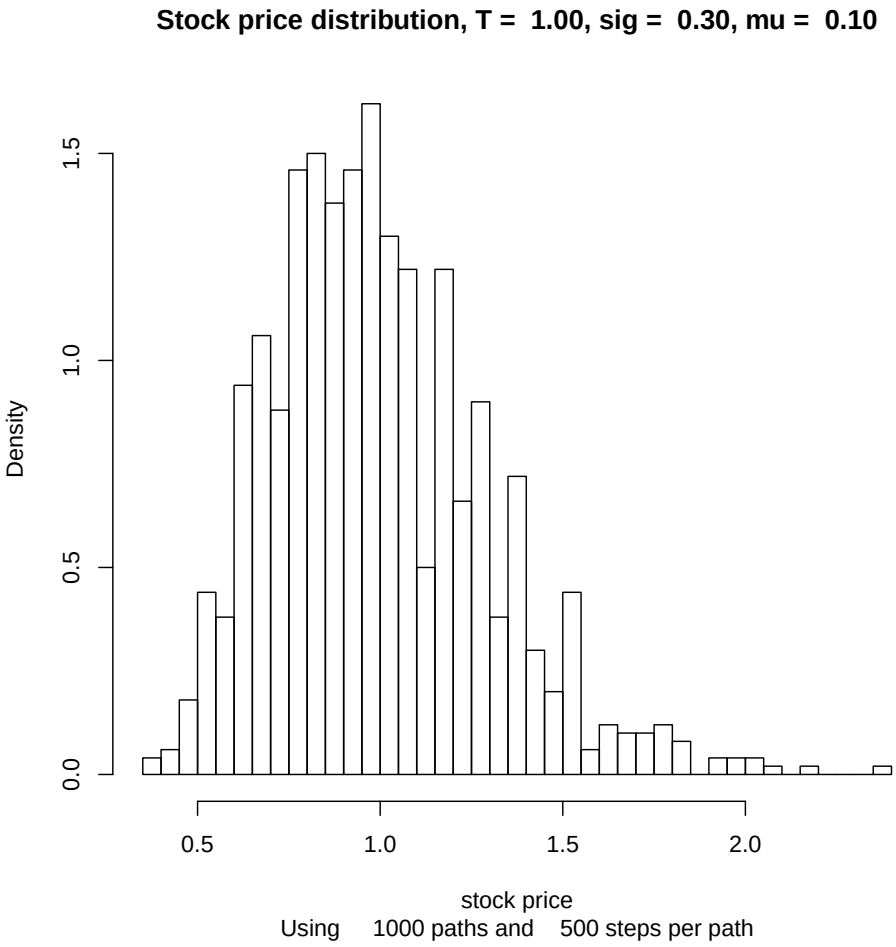

Figure 9: Histogram of $m =$ one thousand realizations of S_T.

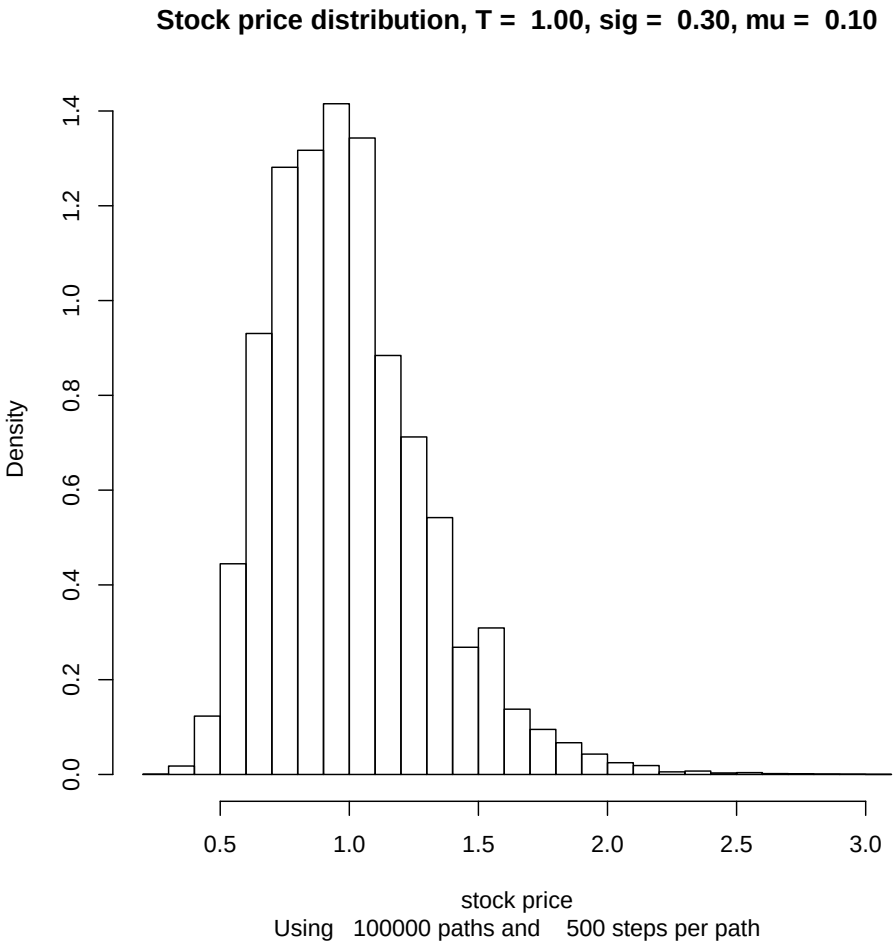

Figure 10: Histogram of $m =$ one hundred thousand realizations of S_T.

binHistogram.R

```r
#    create a histogram of the price distribution at the end
#    of a binomial tree process

n   = 500       # number of steps in a path
mu  = .1        # expected rate of return
sig = .3        # volatility
T   = 1         # final time
S0  = 1
m   = 100000     # number of paths

S   = 1:m       # will hold the final prices
for ( j in 1:m){
    p = binPath(n, mu, sig, S0, T)
    S[j] = p[n+1]
  }

title    = sprintf("Stock price distribution, T = %5.2f, sig = %5.2f, mu = %5.2f",
                                              T,         sig,         mu)
subtitle = sprintf("Using %8d paths and %6d steps per path", m, n)

#------------ copy from here ------------------------------------------------

hist( S,
      breaks = 30,
      probability = TRUE,
      plot        = TRUE,
      main        = title,
      sub         = subtitle,
      xlab        = "stock price")

#------------- copy to here -------------------------------------------------


#               make a file with an image of the plot

pdf("StockPriceHist.pdf")    # tell R to create a .pdf file for the plot

#     Plot commands, copy and paste from above

#------------ copy from here ------------------------------------------------

hist( S,
      breaks = 30,
      probability = TRUE,
      plot        = TRUE,
      main        = title,
      sub         = subtitle,
      xlab        = "stock price")

#------------- copy to here -------------------------------------------------
```

Figure 11: Script that makes the histograms of Figure 9 and 10