Option Pricing Functions to Accompany *Derivatives Markets*

Robert McDonald

June 27, 2016

Contents

1 Introduction

This vignette is an overview to the functions in the *derivmkts* package, which was conceived as a companion to my book *Derivatives Markets* (McDonald, 2013). The material has an educational focus. There are other option pricing packages for R, but this package has several distinguishing features:

- function names (mostly) correspond to those in *Derivatives Markets*.
- vectorized Greek calculations are convenient both for individual options and for portfolios
- the `quincunx` function illustrates the workings of a quincunx (Galton board).
- binomial functions include a plotting function that provides a visual depiction of early exercise

2 European Calls and Puts

Table 1 lists the Black-Scholes related functions in the package. The functions `bscall`, `bsput`, and `bsopt` provide basic pricing of European calls and puts. There are also options with binary payoffs: cash-or-nothing and asset-or-nothing options. All of these functions are vectorized. The function `bsopt` by default provides option greeks. Here are some examples:

```
s <- 100; k <- 100; r <- 0.08; v <- 0.30; tt <- 2; d <- 0
bscall(s, k, v, r, tt, d)

[1] 24.02

bsput(s, c(95, 100, 105), v, r, tt, d)

[1]  7.488  9.239 11.188
```

3 Barrier Options

There are pricing functions for the following barrier options:

- down-and-in and down-and-out barrier binary options
- up-and-in and up-and-out barrier binary options
- more standard down- and up- calls and puts, constructed using the barrier binary options

Naming for the barrier options generally follows the convention

Function	Description
bscall	European call
bsput	European put
bsopt	European call and put and associated Greeks: delta, gamma, vega, theta, rho, psi, and elasticity
assetcall	Asset-or-nothing call
assetput	Asset-or-nothing put
cashcall	Cash-or-nothing call
cashput	Cash-or-nothing put

Table 1: Black-Scholes related option pricing functions

```
[u|d][i|o][call|put]
```

which means that the option is "up" or "down", "in" or "out", and a call or put.[1] An up-and-in call, for example, would be denoted by `uicall`. For binary options, we add the underlying, which is either the asset or $1: cash:

```
[asset|cash][u|d][i|o][call|put]
```

```
H <- 115
bscall(s, c(80, 100, 120), v, r, tt, d)

[1] 35.28 24.02 15.88

uicall(s, c(80, 100, 120), v, r, tt, d, H)

[1] 34.55 23.97 15.88

bsput(s, c(80, 100, 120), v, r, tt, d)

[1]  3.450  9.239 18.141

uoput(s, c(80, 100, 120), v, r, tt, d, H)

[1] 2.328 5.390 9.070
```

4 Perpetual American Options

The functions `callperpetual` and `putperetual` price infinitely-lived American options. The pricing formula assumes that all inputs (risk-free rate, volatility, dividend yield) are fixed. This is of course usual with the basic option pricing formulas, but it is more of a conceptual stretch for an infinitely-lived option than for a 3-month option.

[1] This naming convention differs from that in *Derivatives Markets*, in which names are `callupin`, `callupout`, etc. Thus, I have made both names are available for these functions.

In order for the option to have a determined value, the dividend yield on the underlying asset must be positive if the option is a call. If this is not true, the call is never exercised and the price is undefined.[2] Similarly, the risk-free rate must be positive if the option is a put.

By default, the perpetual pricing formulas return the price. By setting `showbarrier=TRUE`, the function returns both the option price and the stock price at which the option is optimally exercised (the "barrier"). Here are some examples:

```
s <- 100; k <- 100; r <- 0.08; v <- 0.30; tt <- 2; d <- 0.04
callperpetual(s, c(95, 100, 105), v, r, d)

[1] 44.71 43.82 43.00

callperpetual(s, c(95, 100, 105), v, r, d, showbarrier=TRUE)

$price
[1] 44.71 43.82 43.00

$barrier
[1] 338.6 356.4 374.2
```

5 Option Greeks

Greeks for vanilla and barrier options can be computed using the `greeks` function, which is a wrapper for any pricing function that returns the option price and which uses the default naming of inputs.[3]

```
H <- 105
greeks(uicall(s, k, v, r, tt, d, H))

              uicall
Price      18.719815
Delta       0.605436
Gamma       0.008011
Vega        0.480722
Rho         0.836133
Theta      -0.012408
Psi        -1.210530
Elasticity  3.234200
```

[2] A well-known result (Merton, 1973) is that a standard American call is never exercised before expiration if the dividend yield is zero and the interest rate is non-negative. A perpetual call with $\delta = 0$ and $r > 0$ would thus never be exercised. The limit of the option price as $\delta \to 0$ is s, so in this case the function returns the stock price as the option value.

[3] In this version of the package, I have two alternative functions that return Greeks:

- The `bsopt` function by default produces prices and Greeks for European calls and puts.
- The `greeks2` function takes as arguments the name of the pricing function and then inputs as a list.

These may be deprecated in the future. `greeks2` is more cumbersome to use but may be more robust. I welcome feedback on these functions and what you find useful.

The value of this approach is that you can easily compute Greeks for spreads and custom pricing functions. Here are two examples. First, the value at time 0 of a prepaid contract that pays S_T^a at time T is given by the `powercontract()` function:

```
powercontract <- function(s, v, r, tt, d, a) {
    price <- exp(-r*tt)*s^a* exp((a*(r-d) + 1/2*a*(a-1)*v^2)*tt)
}
```

We can easily compute the Greeks for a power contract:

```
greeks(powercontract(s=40, v=.08, r=0.08, tt=0.25, d=0, a=2))

           powercontract
Price           1634.936
Delta             81.747
Gamma              2.044
Vega               0.654
Rho                4.087
Theta             -0.387
Psi               -8.175
Elasticity         2.000
```

Second, consider a bull spread in which we buy a call with a strike of k_1 and sell a call with a strike of k_2. We can create a function that computes the value of the spread, and then compute the greeks for the spread by using this newly-created function together with `greeks()`:

```
bullspread <- function(s, v, r, tt, d, k1, k2) {
    bscall(s, k1, v, r, tt, d) - bscall(s, k2, v, r, tt, d)
}
greeks(bullspread(39:41, .3, .08, 1, 0, k1=40, k2=45))

           bullspread_39 bullspread_40 bullspread_41
Price          2.0020318     2.1551927     2.306e+00
Delta          0.1542148     0.1519426     1.487e-01
Gamma         -0.0017692    -0.0027545    -3.614e-03
Vega          -0.0080732    -0.0132218    -1.822e-02
Rho            0.0401235     0.0392251     3.793e-02
Theta         -0.0005476    -0.0003164    -8.246e-05
Psi           -0.0601438    -0.0607771    -6.099e-02
Elasticity     3.0041376     2.8200287     2.645e+00
```

The Greeks function is vectorized, so you can create vectors of greek values with a single call. This example plots, for a bull spread, the gamma as a function of the stock price; see Figure 1.

```
sseq <- seq(1, 100, by=0.5)
x <- greeks(bullspread(sseq, .3, .08, 1, 0, k1=40, k2=45))
plot(sseq, x['Gamma',], type='l')
```

This code produces the plots in Figure 2:

```
k <- 100; r <- 0.08; v <- 0.30; tt <- 2; d <- 0
S <- seq(.5, 250, by=.5)
```

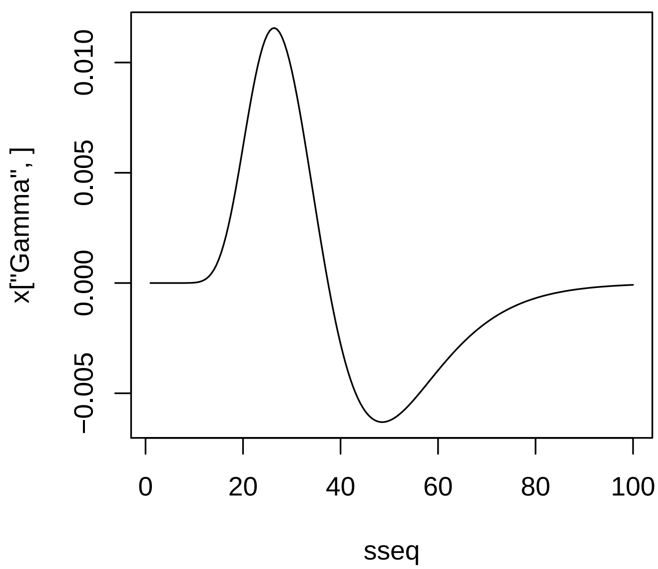

Figure 1: Gamma for a 40-45 bull spread.

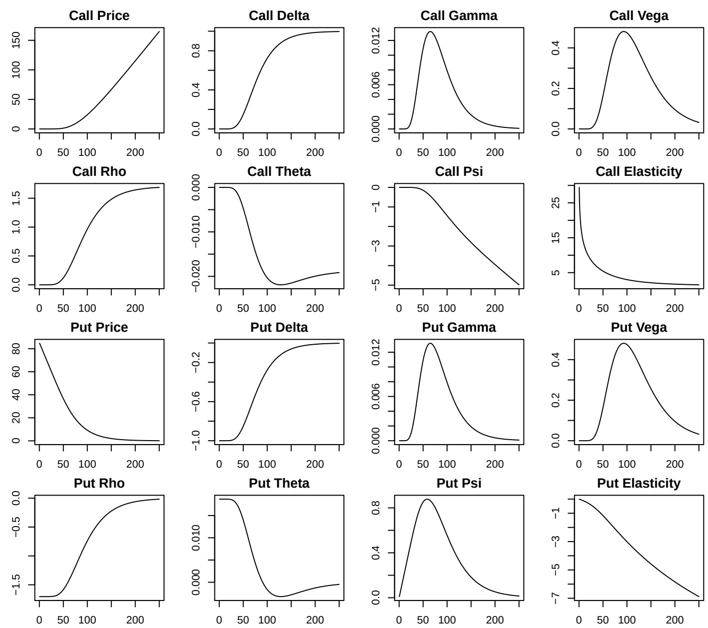

Figure 2: All option Greeks, computed using the greeks() function

```r
Call <- greeks(bscall(S, k, v, r, tt, d))
Put <- greeks(bsput(S, k, v, r, tt, d))
y <- list(Call=Call, Put=Put)
par(mfrow=c(4, 4))  ## create a 4x4 plot
par(mar=c(2,2,2,2))
for (i in names(y)) {
    for (j in rownames(y[[i]])) {  ## loop over greeks
        plot(S, y[[i]][j, ], main=paste(i, j), ylab=j, type='l')
    }
}
```

6 Binomial Pricing of European and American Options

There are two functions related to binomial option pricing:

binomopt computes prices of American and European calls and puts. The function has three optional parameters that control output:

- `returnparams=TRUE` will return as a vector the option pricing inputs, computed parameters, and risk-neutral probability.
- `returngreeks=TRUE` will return as a vector the price, delta, gamma, and theta at the initial node.

- `returntrees=TRUE` will return as a list the price, greeks, the full stock price tree, the exercise status (`TRUE` or `FALSE`) at each node, and the replicating portfolio at each node.

binomplot displays the asset price tree, the corresponding probability of being at each node, and whether or not the option is exercised at each node. This function is described in more detail in Section 10.2.

Here are examples of pricing, illustrating the default of just returning the price, and the ability to return the price plus parameters, as well as the price, the parameters, and various trees:

```
s <- 100; k <- 100; r <- 0.08; v <- 0.30; tt <- 2; d <- 0.03
binomopt(s, k, v, r, tt, d, nstep=3)

price
 20.8

binomopt(s, k, v, r, tt, d, nstep=3, returnparams=TRUE)

   price        s        k        v        r       tt        d    nstep
 20.7961 100.0000 100.0000   0.3000   0.0800   2.0000   0.0300   3.0000
       p       up       dn        h
  0.4391   1.3209   0.8093   0.6667

binomopt(s, k, v, r, tt, d, nstep=3, putopt=TRUE)

price
12.94

binomopt(s, k, v, r, tt, d, nstep=3, returntrees=TRUE, putopt=TRUE)

$price
price
12.94

$greeks
    delta     gamma     theta
-0.335722  0.010614 -0.007599

$params
       s        k        v        r       tt        d    nstep        p
100.0000 100.0000   0.3000   0.0800   2.0000   0.0300   3.0000   0.4391
      up       dn        h
  1.3209   0.8093   0.6667

$oppricetree
      [,1]   [,2]   [,3]  [,4]
[1,] 12.94  3.816  0.000  0.00
[2,]  0.00 21.338  7.176  0.00
[3,]  0.00  0.000 34.507 13.49
[4,]  0.00  0.000  0.000 47.00

$stree
     [,1]   [,2]   [,3]   [,4]
[1,]  100 132.09 174.47 230.45
```

```
[2,]    0  80.93 106.89 141.19
[3,]    0   0.00  65.49  86.51
[4,]    0   0.00   0.00  53.00

$probtree
     [,1]   [,2]   [,3]    [,4]
[1,]    1 0.4391 0.1928 0.08464
[2,]    0 0.5609 0.4926 0.32441
[3,]    0 0.0000 0.3146 0.41445
[4,]    0 0.0000 0.0000 0.17650

$exertree
      [,1]  [,2]  [,3]  [,4]
[1,] FALSE FALSE FALSE FALSE
[2,] FALSE FALSE FALSE FALSE
[3,] FALSE FALSE  TRUE  TRUE
[4,] FALSE FALSE FALSE  TRUE

$deltatree
        [,1]    [,2]    [,3]
[1,] -0.3357 -0.1041  0.0000
[2,]  0.0000 -0.6471 -0.2419
[3,]  0.0000  0.0000 -0.9802

$bondtree
      [,1]  [,2]  [,3]
[1,] 46.51 17.56  0.00
[2,]  0.00 73.71 33.03
[3,]  0.00  0.00 94.81
```

7 Asian Options

There are analytical functions for valuing geometric Asian options and Monte Carlo routines for valuing arithmetic Asian options. Be aware that the `greeks()` function at this time will not work automatically with geometric Asian options nor (for different reasons) with arithmetic Asian options. I plan to address this in a future release.[4]

7.1 Geometric Asian Options

Geometric Asian options can be valued using the Black-Scholes formulas for vanilla calls and puts, with modified inputs. The functions return both call and put prices with a named vector:

[4] At this time the `greeks()` function will not work with options valued using Monte Carlo. The reason is that each invocation of the pricing function will start with a different random number seed, resulting in some price variation that is due solely to random variation. Random number generation and setting the seed is a global change. In a future release I hope to address this by saving and restoring the seed within the greeks function. For the curious, a Stackoverflow post discusses this issue.

```
s <- 100; k <- 100; r <- 0.08; v <- 0.30; tt <- 2; d <- 0.03; m <- 3
geomavgpricecall(s, 98:102, v, r, tt, d, m)

[1] 14.01 13.56 13.12 12.70 12.28

geomavgpricecall(s, 98:102, v, r, tt, d, m, cont=TRUE)

[1] 10.952 10.498 10.058  9.632  9.219

geomavgstrikecall(s, k, v, r, tt, d, m)

[1] 9.058
```

7.2 Arithmetic Asian Options

Monte Carlo valuation is used to price arithmetic Asian options. For efficiency, the function `arithasianmc` returns call and put prices for average price and average strike options. By default the number of simulations is 1000. Optionally the function returns the standard deviation of each estimate

```
arithasianmc(s, k, v, r, tt, d, 3, numsim=5000, printsds=TRUE)

             Call    Put sd Call sd Put
Avg Price  14.313  8.034   22.40 11.556
Avg Strike  8.323  5.139   14.53  7.233
Vanilla    20.446 10.983   33.76 15.073
```

The function `arithavgpricecv` uses the control variate method to reduce the variance in the simulation. At the moment this function prices only calls, and returns both the price and the regression coefficient used in the control variate correction:

```
arithavgpricecv(s, k, v, r, tt, d, 3, numsim=5000)

price  beta
13.98  1.04
```

8 Jumps and Stochastic Volatility

The `mertonjump` function returns call and put prices for a stock that can jump discretely. A poisson process controls the occurrence of a jump and the size of the jump is lognormally distributed. The parameter `lambda` is the mean number of jumps per year, the parameter `alphaj` is the log of the expected jump, and `sigmaj` is the standard deviation of the log of the jump. The jump amount is thus drawn from the distribution

$$Y \sim \mathcal{N}(\alpha_J - 0.5\sigma_J^2, \sigma_J^2)$$

```
mertonjump(s, k, v, r, tt, d, lambda=0.5, alphaj=-0.2, vj=0.3)

 Call   Put
23.98 15.02

c(bscall(s, k, v, r, tt, d), bsput(s, k, v, r, tt, d))

[1] 19.96 11.00
```

9 Bonds

The simple bond functions provided in this version compute the present value of cash flows (`bondpv`), the IRR of the bond (`bondyield`), Macaulay duration (`duration`), and convexity (`convexity`).

```
coupon <- 8; mat <- 20; yield <- 0.06; principal <- 100;
modified <- FALSE; freq <- 2
price <- bondpv(coupon, mat, yield, principal, freq)
price

[1] 123.1

bondyield(price, coupon, mat, principal, freq)

[1] 0.06

duration(price, coupon, mat, principal, freq, modified)

[1] 11.23

convexity(price, coupon, mat, principal, freq)

[1] 170.3
```

10 Functions with Graphical Output

Several functions provide visual illustrations of some aspects of the material.

10.1 Quincunx or Galton Board

The quincunx is a physical device the illustrates the central limit theorem. A ball rolls down a pegboard and strikes a peg, falling randomly either to the left or right. As it continues down the board it continues to strike a series of pegs, randomly falling left or right at each. The balls collect in bins and create an approximate normal distribution.

The quincunx function allows the user to simulate a quincunx, observing the path of each ball and watching the height of each bin as the balls accumulate.

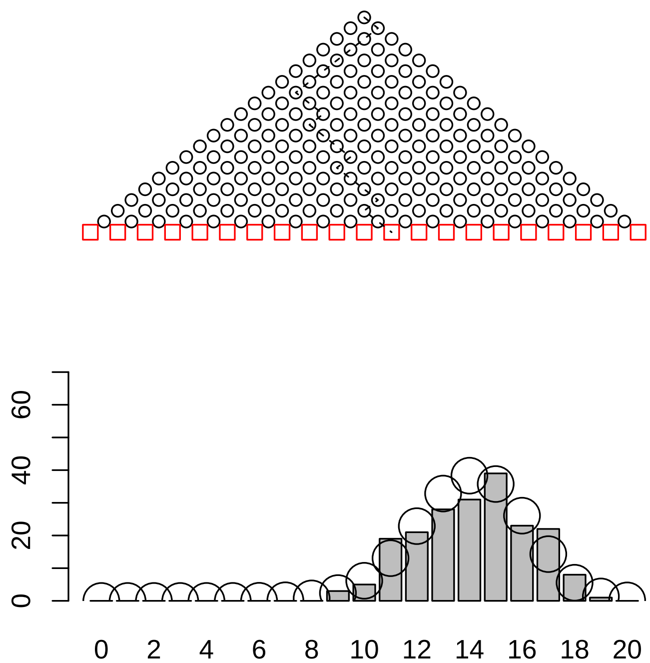

Figure 3: Output from the Quincunx function

More interestingly, the quincunx function permits altering the probability that the ball will fall to the right.

Figure 3 illustrates the function after dropping 200 balls down 20 levels of pegs with a 70% probability that each ball will fall right:

```
par(mar=c(2,2,2,2))
quincunx(n=20, numballs=200, delay=0, probright=0.7)
```

10.2 Plotting the Solution to the Binomial Pricing Model

The `binomplot` function calls `binomopt` to compute the option price and the various trees, which it then uses in plotting:

The first plot, figure 4, is basic:

```
binomplot(s, k, v, r, tt, d, nstep=6, american=TRUE, putopt=TRUE)
```

The second plot, figure 5, adds a display of stock prices and arrows connecting the nodes.

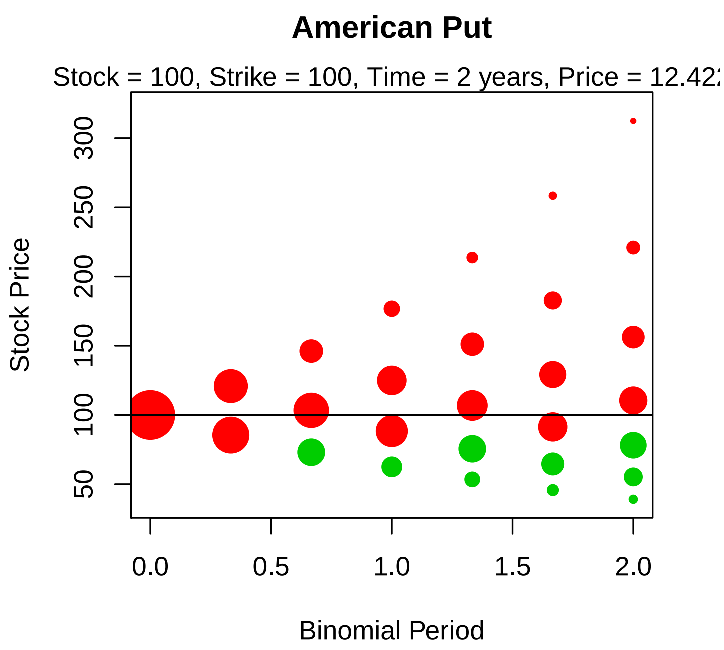

Figure 4: Basic option plot showing stock prices and nodes at which the option is exercised.

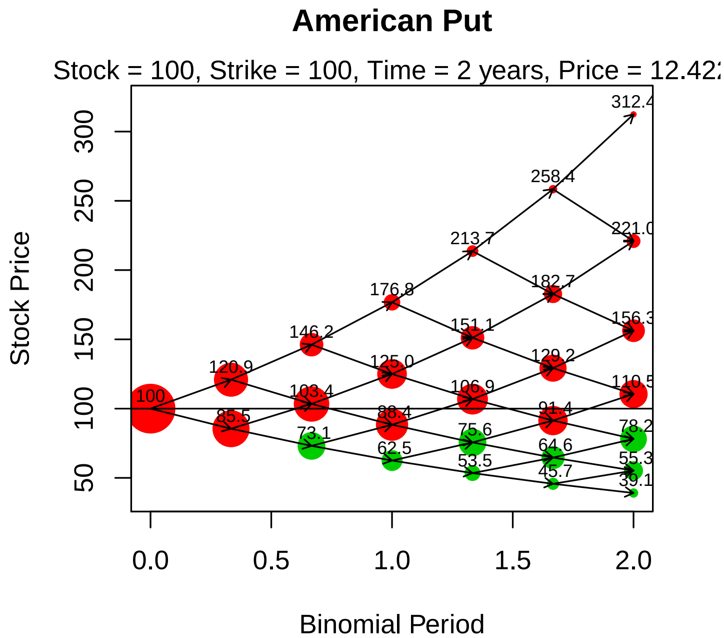

Figure 5: Same plot as Figure 4 except that values and arrows are added to the plot.

```
binomplot(s, k, v, r, tt, d, nstep=6, american=TRUE, putopt=TRUE,
    plotvalues=TRUE, plotarrows=TRUE)
```

As a final example, consider an American call when the dividend yield is positive and `nstep` has a larger value. Figure 6 shows the plot, with early exercise evident.

```
d <- 0.06
binomplot(s, k, v, r, tt, d, nstep=40, american=TRUE)
```

The large value of `nstep` creates a high maximum terminal stock price, which makes details hard to discern in the boundary region where exercise first occurs. We can zoom in on that region by selecting values for `ylimval`; the result is in Figure 7.

```
d <- 0.06
binomplot(s, k, v, r, tt, d, nstep=40, american=TRUE, ylimval=c(75, 225))
```

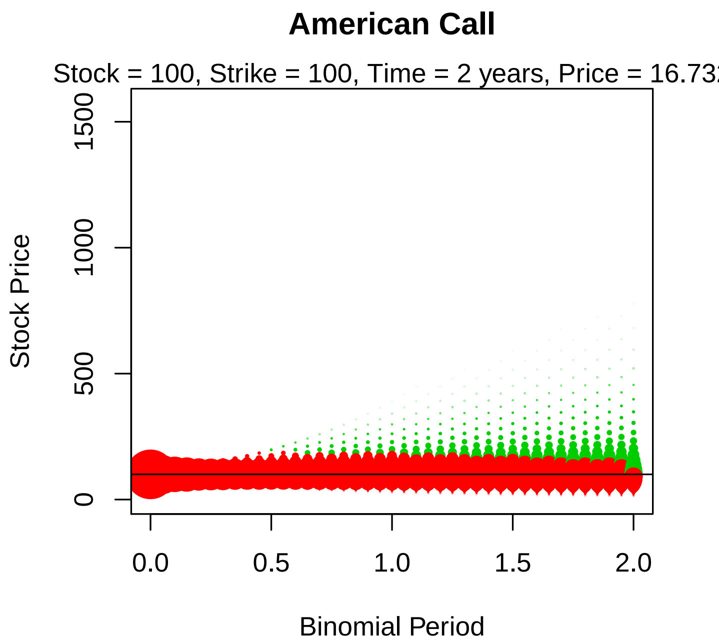

Figure 6: Binomial plot when nstep is 40.

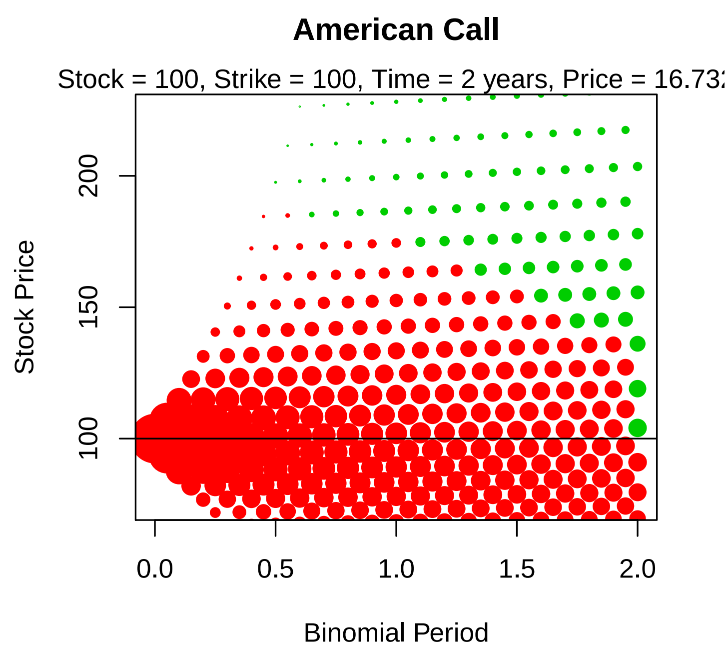

Figure 7: Binomial plot when nstep is 40 using the argument ylimval to focus on a subset.

A Vectorization

Where possible, I have tried to make sure that the pricing functions return vectors. This is automatic in many cases (for example, with the Black-Scholes formula), but there are situations in which achieving robust vectorization requires care when constructing a function. The purpose of this section is to explain the problems I encountered and the solutions I considered. Perhaps I overlooked the obvious or I am ignorant of some details of R. In either case I hope you will let me know! Otherwise, I hope this discussion is helpful to others.

A.1 Automatic Vectorization

```
f = function(a, b, k) a*b + k
f(3, 5, 1)

[1] 16

f(1:5, 5, 1)

[1]  6 11 16 21 26

f(1:6, 1:2, 1)

[1]  2  5  4  9  6 13
```

In this example, R automatically vectorizes the multiplication using the recycling rule. It's worth noting that the third example, in which both arguments are vectorized, but with different length vectors, is an unusual programming construct.[5] This property makes it trivial to perform what-if calculations for an option pricing formula.

A.2 Limitations of Automatic Vectorization

A problem with automatic vectorization occurs when there are conditional statements. With barrier options, for example, it is necessary to check whether the asset price is past the barrier. R's `if` statement is not vectorized, and the `ifelse` function returns output that has the dimension of the conditional.

```
cond1 <- function(a, b, k) {
    if (a > b) {
        a*b + k
    } else {
        k
    }
}
cond1(5, 3, 1)

[1] 16
```

[5]You can produce the same output in python using the itertools module.

```
cond1(5, 7, 1)

[1] 1

cond1(3:7, 5, 1)

Warning in if (a > b) {:  the condition has length > 1 and only the first element will
be used

[1] 1
```

The third invocation of `cond1` causes an error because the `if` statement is not vectorized. This can be fixed by rewriting the conditional using `ifelse`, which is vectorized. The following examples all compute correctly because if either a or b are vectors, the conditional statement is vectorized:

```
cond2 <- function(a, b, k) {
    ifelse(a > b,
           a*b + k,
           k
           )
}
cond2(5, 3, 1)

[1] 16

cond2(5, 7, 1)

[1] 1

cond2(3:7, 5, 1)

[1]  1  1  1 31 36
```

There will, however, be a problem if only k is a vector. Suppose we set $a = 5$, $b = 7$, $k = 1:3$. Because $a < b$, we want to produce the output $1, 2, 3$. The following example does not work as desired because neither of the variables in the conditional (a and b) are a vector. Thus the calculation is not vectorized:

```
cond2(5, 7, 1:3)

[1] 1
```

The `ifelse` function returns output with the dimension of the conditional expression, which in this case is a vector of length 1.

A.3 Three Solutions

One solution is to write the function so as to vectorize all the inputs to match the vector length of the longest input. There are at least three ways to do this.

A.3.1 Use a Booleans in Place of `ifelse`

We can create a boolean variable that is true if $a > b$. This will then control which expression is returned:

```
cond2b <- function(a, b, k) {
    agtb <- (a > b)
    agtb*(a*b + k) + (1-agtb)*k
}

cond2b(5, 3, 1)

[1] 16

cond2b(5, 7, 1)

[1] 1

cond2b(3:7, 5, 1)

[1]  1  1  1 31 36

cond2b(5, 7, 1:3)

[1] 1 2 3
```

Whether this solution works in other functions depends on the structure of the calculation and the nature of the output. In particular, if the value of a boolean controls the data structure the function returns (a vector vs a list, for example), then this solution does not work.

A.3.2 Create a Data Frame

We can enforce the recycling rule for all variables by creating a data frame consisting of the inputs and assigning the columns back to the original variables:

```
cond2c <- function(a, b, k) {
    tmp <- data.frame(a, b, k)
    for (i in names(tmp)) assign(i, tmp[[i]])
    ifelse(a > b,
           a*b + k,
           k
           )
}

cond2c(5, 3, 1)

[1] 16

cond2c(5, 7, 1)

[1] 1

cond2c(3:7, 5, 1)
```

```
[1]  1  1  1 31 36
```

```r
cond2c(5, 7, 1:3)
```

```
[1] 1 2 3
```

One drawback of this solution is that we have to be careful to update the `data.frame()` definition within the function if we change the function inputs. It may be easy to overlook this when editing the function. The next solution is a more robust version of the same idea.

A.3.3 Create a Vectorization Function

A final alternative is to create a vectorization function that exploits R's functional capabilities and does not require modifications if the function definition changes. This approach can become quite complicated, but is relatively easy to understand in simple cases. We create a `vectorizeinputs()` function that creates a data frame and vectorizes all variables:

```r
vectorizeinputs <- function(e) {
    ## e is the result of match.call() in the calling function
    e[[1]] <- NULL
    e <- as.data.frame(as.list(e))
    for (i in names(e)) assign(i, eval(e[[i]]),
                               envir=parent.frame())
}
```

This function assumes that `match.call()` has been invoked in the calling function. The result of that invocation is manipulated to provide information about the parameters passed to the function and used to create the data frame and pass the variables back to the calling function.

```r
cond3 <- function(a, b, k) {
    vectorizeinputs(match.call())
    ifelse(a > b, a*b + k, k)
}
cond3(5, 7, 1:3)
```

```
[1] 1 2 3
```

```r
cond3(3:7, 5, 1)
```

```
[1]  1  1  1 31 36
```

```r
cond3(3:7, 5, 1:5)
```

```
[1]  1  2  3 34 40
```

```r
cond3(k=1:5, 3:7, 5)
```

```
[1]  1  2  3 34 40
```

This approach becomes more complicated if there are implicit parameters in the function. If truly implicit, these will not be available via `match.call()`, but they can affect the solution. Here is an example:

```
cond4 <- function(a, b, k, multby2=TRUE) {
    vectorizeinputs(match.call())
    ifelse(multby2,
           2*(a*b + k),
           a*b + k
           )
}
cond4(5, 7, 1:3)

[1] 72

cond4(3:7, 5, 1)

[1] 32

cond4(3:7, 5, 1:5)

[1] 32

cond4(k=1:5, 3:7, 5)

[1] 32
```

The output is not vectorized because the implicit parameter `multby2` is implicit — it is not explicit in the function call — and therefore it is not vectorized. One way to handle this case is to rewrite the `vectorizeinputs` function to retrieve the full set of function inputs for the called function. The name of the function is available through `match.call()[[1]]`, and the function parameters are available using the `formals` function. We can then add the implicit parameters to the vectorized set of inputs. The function `vectorizeinputs2` takes this approach:

```
vectorizeinputs2 <- function(e) {
    funcname <- e[[1]]
    fvals <- formals(eval(funcname))
    fnames <- names(fvals)
    e[[1]] <- NULL
    e <- as.data.frame(as.list(e))
    implicit <- setdiff(fnames, names(e))
    if (length(implicit) > 0) e <- data.frame(e, fvals[implicit])
    for (i in names(e)) assign(i, eval(e[[i]]),
                               envir=parent.frame())
}

cond5 <- function(a, b, k, multby2=TRUE, altmult=1) {
    vectorizeinputs2(match.call())
    ifelse(multby2,
           2*(a*b + k),
           altmult*(a*b + k)
           )
}
cond5(5, 7, 1:3)
```

```
[1] 72 74 76

cond5(3:7, 5, 1)

[1] 32 42 52 62 72

cond5(3:7, 5, 1:5)

[1] 32 44 56 68 80

cond5(k=1:5, 3:7, 5)

[1] 32 44 56 68 80

cond5(k=1:5, 3:7, 5, multby2=FALSE)

[1] 16 22 28 34 40

cond5(k=1:5, 3:7, 5, multby2=FALSE, altmult=5)

[1]  80 110 140 170 200
```

A.3.4 Why Worry About Vectorization?

R provides looping constructs and `apply` functions. It might seem that it's not necessary to worry about vectorization. There are two reasons that I choose to vectorize where feasible.

First, I personally find vectorization convenient and transparent. I find vectorized code easier to read and it fits the way I like to work. This is an aesthetic argument.

Second, in my experience the apply functions are a real hurdle for new R users. Automatic vectorization makes it possible to perform complicated calculations in a straightforward and intuitive way.

B Bibliography

References

McDonald, R. L., 2013, *Derivatives Markets*, Pearson/Addison Wesley, Boston, MA, 3rd edition.

Merton, R. C., 1973, "Theory of Rational Option Pricing," *Bell Journal of Economics and Management Science*, 4(1), 141–183.