Lecture 12: Functions as Objects, and Intro to Optimization

Statistical Computing, 36-350

Monday October 26, 2015

Previously

- Writing our own functions
- Dividing labor with multiple functions
- Refactoring to create higher-level operations
- Using `apply`, `sapply`, etc., to avoid iteration

Outline

- Functions are objects, and can be arguments to other functions
- Functions are objects, and can be returned by other functions
- Example: `surface`

Functions as objects

- In R, functions are objects, just like everything else!
- This means that they can be passed to functions as arguments and returned by functions as outputs as well

Functions of functions: computationally

- We often want to do very similar things to many different functions
- The procedure is the same, only the function we're working with changes
- Thus, write one function to do the job, and pass the function as an argument
- Because R treats a function like any other object, we can do this simply: invoke the function by its argument name in the body
- We have already seen examples

Functions that take functions as arguments

Some examples we've seen:

- `apply()`, `sapply()`, etc.: take *this* function and use it on all of *these* objects
- `nlm()`: Take *this* function and try to make it small, starting from *here*
- `curve()`: Evaluate *this* function over *that* range, and plot the results

Peeking at a function's definition

Typing a function's name, without parentheses, in the terminal gives you its source code:

```
sample
```

```
## function (x, size, replace = FALSE, prob = NULL)
## {
##     if (length(x) == 1L && is.numeric(x) && x >= 1) {
##         if (missing(size))
##             size <- x
##         sample.int(x, size, replace, prob)
##     }
##     else {
##         if (missing(size))
##             size <- length(x)
##         x[sample.int(length(x), size, replace, prob)]
##     }
## }
## <bytecode: 0x104054e40>
## <environment: namespace:base>
```

This isn't always that explicit, because some functions are defined in a lower level language (like C or Fortran)

```
log
```

```
## function (x, base = exp(1))  .Primitive("log")
```

```
rowSums
```

```
## function (x, na.rm = FALSE, dims = 1L)
## {
##     if (is.data.frame(x))
##         x <- as.matrix(x)
##     if (!is.array(x) || length(dn <- dim(x)) < 2L)
##         stop("'x' must be an array of at least two dimensions")
##     if (dims < 1L || dims > length(dn) - 1L)
##         stop("invalid 'dims'")
##     p <- prod(dn[-(id <- seq_len(dims))])
##     dn <- dn[id]
##     z <- if (is.complex(x))
##         .Internal(rowSums(Re(x), prod(dn), p, na.rm)) + (0+1i) *
##             .Internal(rowSums(Im(x), prod(dn), p, na.rm))
##     else .Internal(rowSums(x, prod(dn), p, na.rm))
##     if (length(dn) > 1L) {
##         dim(z) <- dn
##         dimnames(z) <- dimnames(x)[id]
##     }
```

```
##     else names(z) <- dimnames(x)[[1L]]
##     z
## }
## <bytecode: 0x104cfdbf8>
## <environment: namespace:base>
```

The function class

Functions are their own **class** in R:

```
class(sin)
```

```
## [1] "function"
```

```
class(sample)
```

```
## [1] "function"
```

```
resample = function(x) { sample(x, size=length(x), replace=TRUE) }
class(resample)
```

```
## [1] "function"
```

Some facts about functions

- Functions can be put into lists or even arrays
- A call to `function()` creates and returns a function object
  - Can see the body `body()`
  - Can see the arguments with `args()`
  - Can see the environment with `environment()`

```
body(resample)
```

```
## {
##     sample(x, size = length(x), replace = TRUE)
## }
```

```
args(resample)
```

```
## function (x)
## NULL
```

```
environment(resample)
```

```
## <environment: R_GlobalEnv>
```

R has separate **types** for built-in functions and for those written in R:

```
typeof(resample)
```

```
## [1] "closure"
```

```
typeof(sample)
```

```
## [1] "closure"
```

```
typeof(sin)
```

```
## [1] "builtin"
```

- Why `closure` for written-in-R functions? Because expressions are "closed" by referring to the parent environment
- There's also a 2nd class of built-in functions called `primitive`

Anonymous functions

- `function()` returns an object of class `function`
- We usually assign that object to a name
- But if we don't have an assignment, we get an **anonymous function**
- Usually part of some larger expression:

```
sapply((-2):2,function(log.ratio){exp(log.ratio)/(1+exp(log.ratio))})
```

```
## [1] 0.1192029 0.2689414 0.5000000 0.7310586 0.8807971
```

- Anonymous functions are handy when connecting to other pieces of code, especially in things like `apply` and `sapply`
- Won't cluttering the workspace
- But can't be examined or re-used later

Example: `grad()`

- Many problems in statistics come down to optimization (so do lots of problems in economics, physics, computer science, etc.)
- Lots of optimization routines require the gradient of the **objective function**—this is the function that is to be minimized or maximized
- Recall, the gradient of f at $x = (x_1, \ldots x_n)$ is just the vector that collects all the partial derivatives:

$$\nabla f(x) = \begin{pmatrix} \frac{\partial f(x)}{\partial x_1} \\ \vdots \\ \frac{\partial f(x)}{\partial x_n} \end{pmatrix}$$

Note that we do basically the same thing to get the gradient of f at x no matter what f is:

```
Find partial derivative of f with respect to each component of x
Return the vector of partial derivatives
```

- It makes no sense to rewrite this every time we change f!
- Hence, write code to calculate the gradient of an arbitrary function
- We *could* write our own, but there are lots of tricky issues
  - Best way to calculate partial derivative?
  - What if x is at the edge of the domain of f?
- Fortunately, someone has already done this for us

From the package `numDeriv`

```
library(numDeriv)
args(grad)
```

```
## function (func, x, method = "Richardson", side = NULL, method.args = list(),
##     ...)
## NULL
```

- `func` is a function which returns a single floating-point value
- `x` is a vector, at which we want to evaluate the derivative of `func`
- Extra arguments in `...` get passed along to `func`
- Other functions in the package for, e.g., the Hessian matrix (matrix of second partial derivatives)

Simple example

So, does it work as advertized?

```
simpleFun = function(x) {
  return(x[1]^2 + 1/3*x[2]^2)
}
(xpt = runif(n=2,min=-2,max=2))
```

```
## [1] -1.0116002 -0.7158292
```

```
grad(simpleFun, xpt)
```

```
## [1] -2.0232004 -0.4772195
```

```
max(abs(grad(simpleFun,xpt) - c(2*xpt[1],2/3*xpt[2])))
```

```
## [1] 5.519141e-12
```

Complex example

Let's try a more complicated example ...

```
complicatedFun = function(x) {
  return((1/2*x[1]^2-1/4*x[2]^2+3)*cos(2*x[1]+1-exp(x[2])))
}
(xpt = runif(n=2,min=-2,max=2))
```

```
## [1] -0.01966738 -1.34804413
```

```
grad(complicatedFun, xpt)
```

```
## [1] -3.2988141  0.9415997
```

We could differentiate the above by hand (and you should, for practice), but here's another way that saves us work. Have R calculate the derivatives symbolically:

```
(d1 = D(expression((1/2*x^2-1/4*y^2+3)*cos(2*x+1-exp(y))), "x"))
```

```
## 1/2 * (2 * x) * cos(2 * x + 1 - exp(y)) - (1/2 * x^2 - 1/4 *
##     y^2 + 3) * (sin(2 * x + 1 - exp(y)) * 2)
```

```
(d2 = D(expression((1/2*x^2-1/4*y^2+3)*cos(2*x+1-exp(y))), "y"))
```

```
## (1/2 * x^2 - 1/4 * y^2 + 3) * (sin(2 * x + 1 - exp(y)) * exp(y)) -
##     1/4 * (2 * y) * cos(2 * x + 1 - exp(y))
```

```
(ans1 = eval(d1, envir=data.frame(x=xpt[1], y=xpt[2])))
```

```
## [1] -3.298814
```

```
(ans2 = eval(d2, envir=data.frame(x=xpt[1], y=xpt[2])))
```

```
## [1] 0.9415997
```

```
max(abs(grad(complicatedFun, xpt) - c(ans1,ans2)))
```

```
## [1] 5.175127e-10
```

Note: that symbolic calculation using `D()` is much more limited than numerical calculation using `grad()`, i.e., it only works for certain functions that R knows how to differentiate in closed form

Gradient descent

The bread and butter of optimization routines: **gradient descent**. The idea is simple—just calculate the gradient of the function you're trying to (say) minimizing, move in the direction of its negative gradient, and repeat

With our knowledge of `grad()`, we can now write a very useful function for peforming gradient descent

```
grad.descent = function(f, x0, max.iter=200, step.size=0.05,
  stopping.deriv=0.01, ...) {

  n = length(x0)
  xmat = matrix(0,nrow=n,ncol=max.iter)
  xmat[,1] = x0

  for (k in 2:max.iter) {
    # Calculate the gradient
    grad.cur = grad(f,xmat[,k-1],...)

    # Should we stop?
    if (all(abs(grad.cur) < stopping.deriv)) {
      k = k-1; break
    }

    # Move in the opposite direction of the grad
    xmat[,k] = xmat[,k-1] - step.size * grad.cur
  }

  xmat = xmat[,1:k] # Trim
  return(list(x=xmat[,k], xmat=xmat, k=k))
}
```

Let's try it out on our simple example!

```
x0 = c(-1.9,-1.9)
gd = grad.descent(simpleFun,x0)
gd$x
```

```
## [1] -5.437919e-07 -1.490486e-02
```

```
gd$k
```

```
## [1] 144
```

Note: the minimum here is achieved at (0,0), so this is right

---

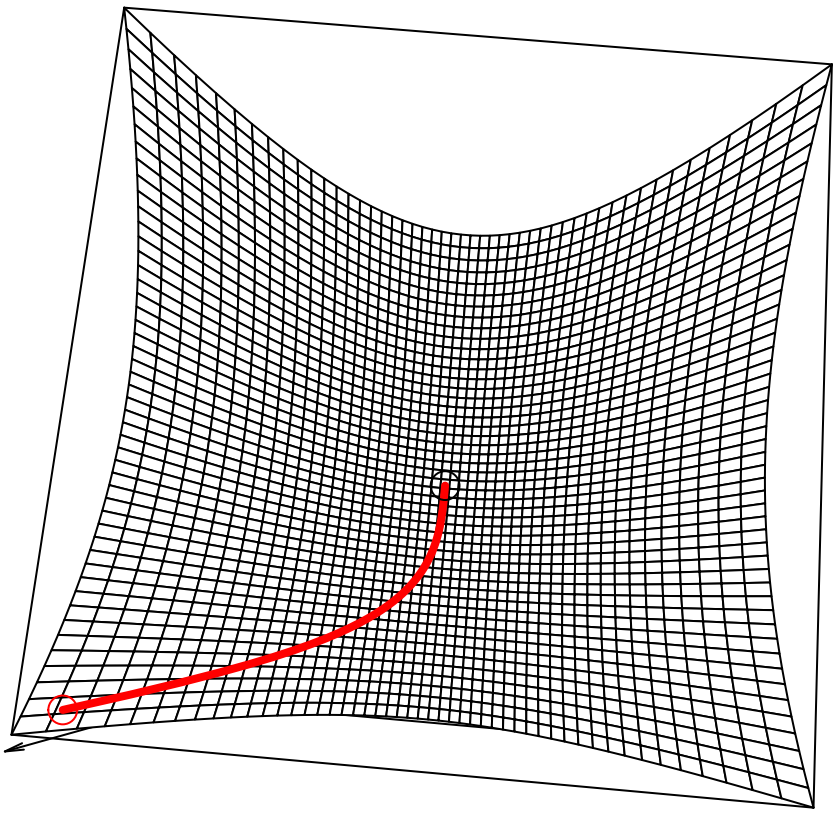
Let's look at the gradient descent path traversed!

```
# Evaluate our function over a grid of (x,y) pairs between -2 and 2
ng = 50
xgr = ygr = seq(-2,2,length=ng)
vals = matrix(0,ng,ng)
for (i in 1:ng)
  for (j in 1:ng)
    vals[i,j] = simpleFun(c(xgr[i],ygr[j]))

# Use the persp function for a nice 3d plot
orig.mar = par()$mar # Save the original margins
par(mar=c(0,0,0,0))  # Make the margins small
r = persp(xgr,ygr,vals,theta=5,phi=80,xlab="",ylab="",zlab="")

# (We'll see cleaner code for these last two steps soon!)

# Draw the gradient descent path on top of this
points(trans3d(x0[1],x0[2],simpleFun(x0),r),col="red",cex=2)
lines(trans3d(gd$xmat[1,],gd$xmat[2,],apply(gd$xmat,2,simpleFun),r),
      lwd=4,col="red")
points(points(trans3d(gd$x[1],gd$x[2],simpleFun(gd$x),r),
              col="black",cex=2))
```



```r
par(mar=orig.mar) # Restore the original margins
```

The power of gradient descent

This is a very broadly applicable algorithm! Works equally well when `f` is:

- least squares loss for an ordinary regression
- Huber loss for a robust regression
- negative log likelihood
- cost of a production plan
- etc.

We'll learn much more next time

Example: gradient descent for linear regression

Let's set up a linear regression simulation

```r
n = 100
p = 2
pred = matrix(rnorm(n*p),n,p)
beta = c(1,4)
resp = pred %*% beta + rnorm(n)
(lm.coefs = coef(lm(resp ~ pred + 0)))
```

```
##     pred1     pred2
## 0.9098495 3.9359503
```

Let's now try out gradient descent:

```
tryCatch({
  out = grad.descent(function(beta) { sum((resp-pred%*%beta)^2)},
                     x0=c(0,0), step.size=0.05, max.iter=200)
  }, error = function(err) { cat(err$message) })
```

```
## function returns NA at 5.48953455200445e+1482.11405987595638e+149 distance from x.
```

Uh oh! What the heck happened??

You should practice your debugging skill to confirm this, but the step size is simply too large, and so gradient descent is not converging

---

A simple fix is just to take a smaller step size

```
out = grad.descent(function(beta) { sum((resp-pred%*%beta)^2)},
                   x0=c(0,0), step.size=1e-3, max.iter=200)

# Accurate to 3 digits in 60 steps, not too shabby!
out$k
```

```
## [1] 56
```

```
out$x
```

```
## [1] 0.9097915 3.9359350
```

```
lm.coefs
```

```
##     pred1     pred2
## 0.9098495 3.9359503
```

We perhaps don't want to fiddle around with the step size manually (what's the problem with this? what's the problem with taking it just to be super tiny, so that we always converge?)

Next time, we'll learn a more principled strategy

curve()

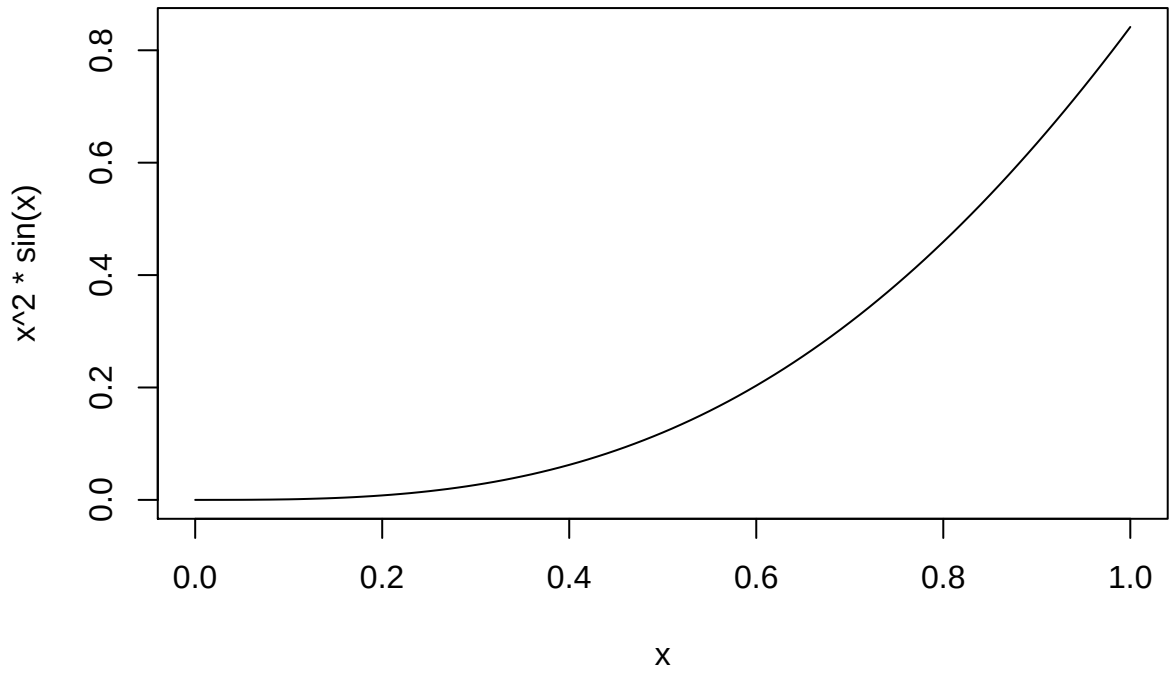
We've seen `curve()` a few times so far. A call to `curve` looks like this:

```
curve(expr, from = a, to = b, ...)
```

Here `expr` is some expression involving a variable called `x`, which is swept `from` the value `a` `to` the value `b`, and `...` are other plotting arguments

For example:

```
out = curve(x^2 * sin(x), 0, 1)
```




```
names(out)
```

```
## [1] "x" "y"
```

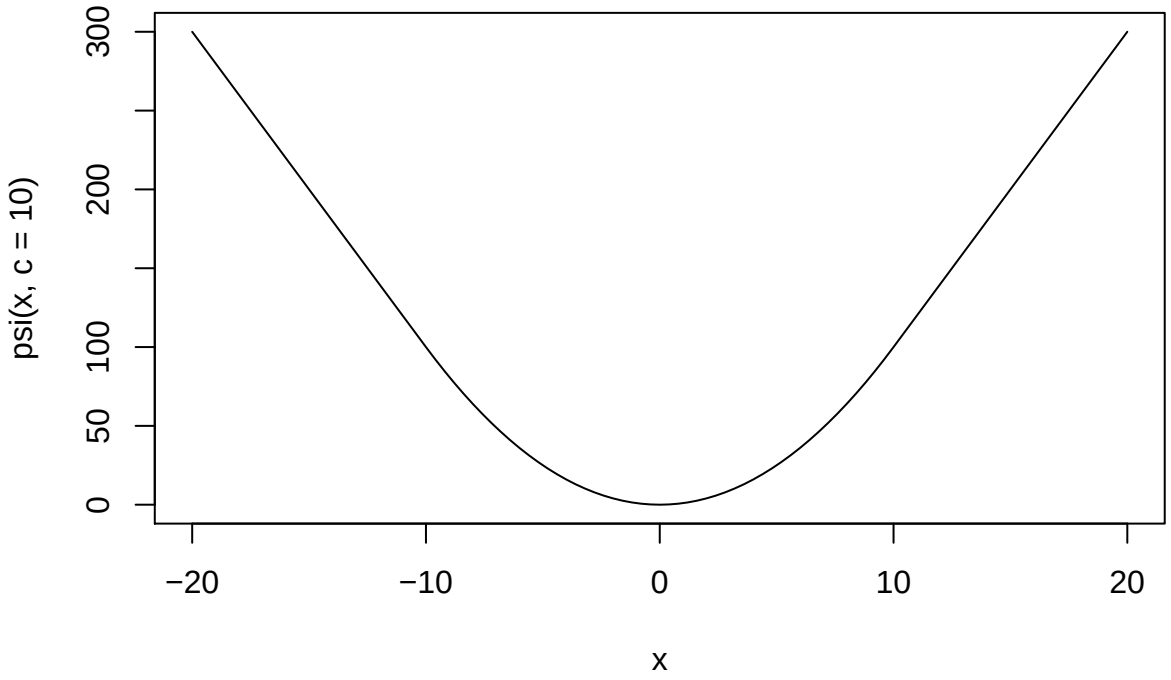
```
head(cbind(out$x,out$y))
```

```
##      [,1]         [,2]
## [1,] 0.00 0.000000e+00
## [2,] 0.01 9.999833e-07
## [3,] 0.02 7.999467e-06
## [4,] 0.03 2.699595e-05
## [5,] 0.04 6.398293e-05
## [6,] 0.05 1.249479e-04
```

Using `curve()` with our own functions

If we have defined a function already, we can use it in `curve`:

```
psi = function(x,c=1) { ifelse(abs(x)>c,2*c*abs(x)-c^2,x^2) }
curve(psi(x,c=10),from=-20,to=20)
```



Problems when our own functions aren't vectorized

If our function doesn't take vectors to vectors, `curve()` becomes unhappy:

```
gmp = read.table("http://www.stat.cmu.edu/~ryantibs/statcomp/lectures/gmp.dat")
gmp$pop = round(gmp$gmp/gmp$pcgmp)
mse = function(y0,a,Y=gmp$pcgmp,N=gmp$pop) { mean((Y - y0*(N^a))^2) }

tryCatch({
  curve(mse(a=x,y0=6611),from=0.10,to=0.15)
  }, error = function(err) { cat(err$message) })
```

```
## Warning in N^a: longer object length is not a multiple of shorter object
## length
```

```
## 'expr' did not evaluate to an object of length 'n'
```

How do we solve this?

Define a new, vectorized function, say with `sapply`:

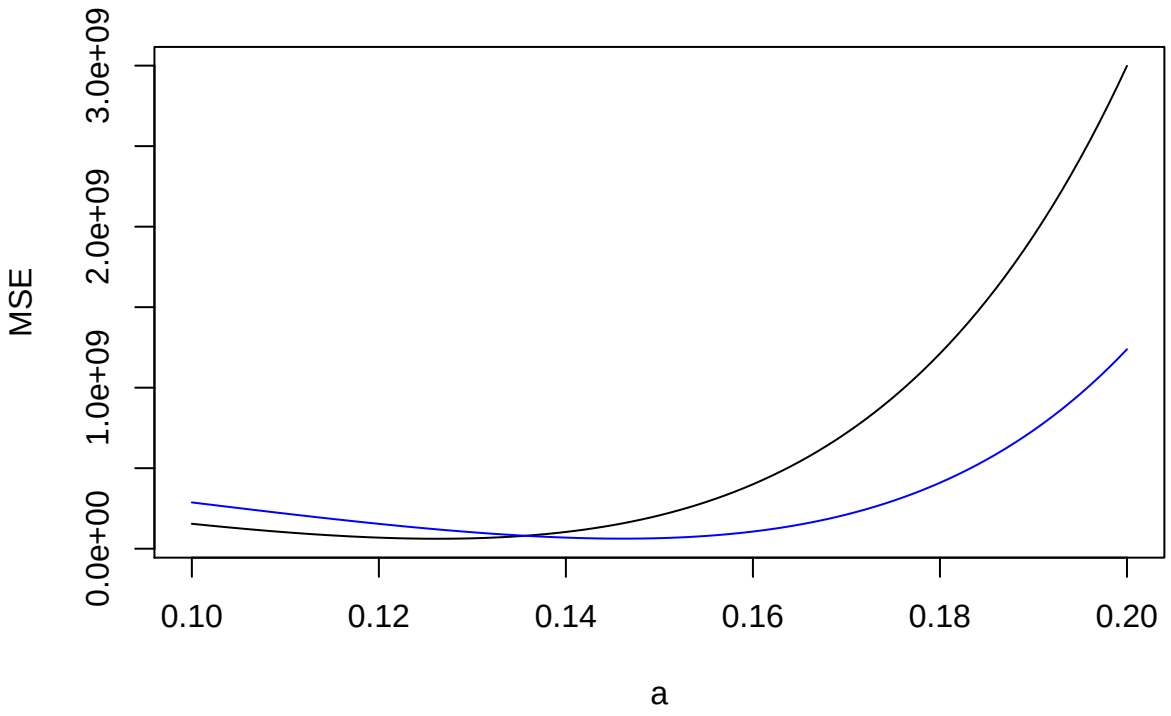
```
sapply(seq(from=0.10,to=0.15,by=0.01),mse,y0=6611)
```

```
## [1] 154701953 102322974  68755654  64529166 104079527 207057513
```

```
mse(6611,0.10)
```

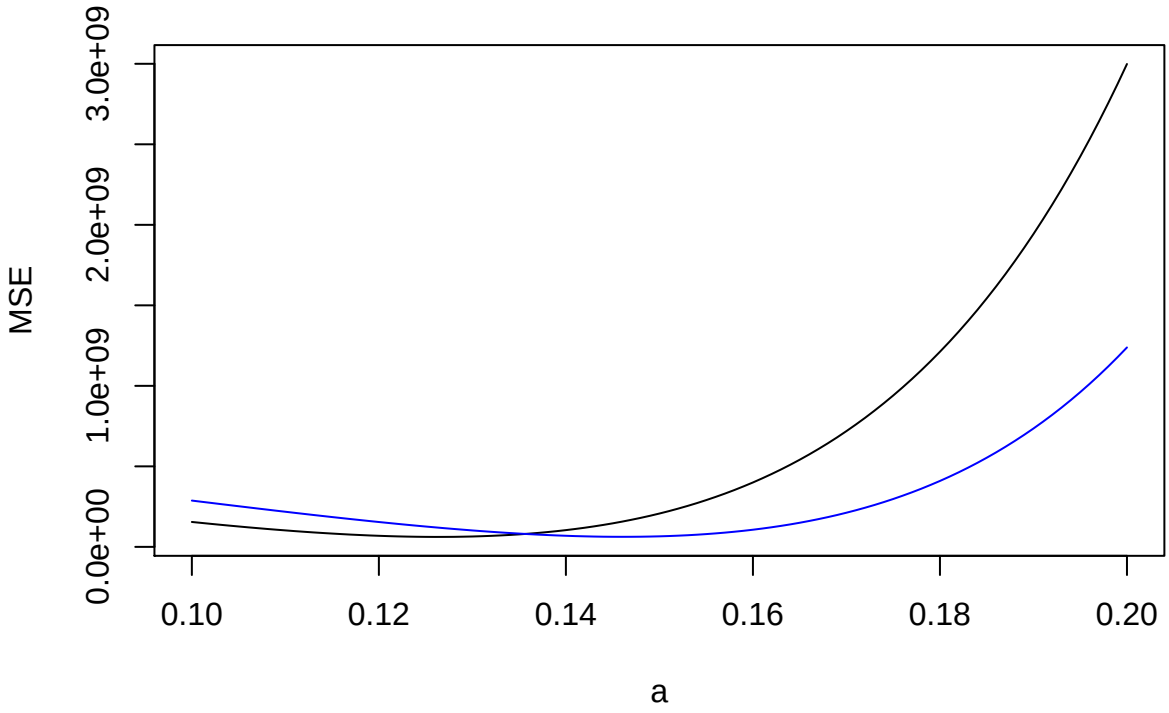
```
## [1] 154701953
```

```
mse.plottable = function(a,...) { return(sapply(a,mse,...)) }
curve(mse.plottable(a=x,y0=6611),from=0.10,to=0.20, xlab="a",ylab="MSE")
curve(mse.plottable(a=x,y0=5100),add=TRUE,col="blue")
```



Alternate strategy: `Vectorize()` returns a new, vectorized function

```
mse.vec = Vectorize(mse, vectorize.args=c("a","y0"))
curve(mse.vec(a=x,y0=6611),from=0.10,to=0.20,xlab="a",ylab="MSE")
curve(mse.vec(a=x,y0=5100),add=TRUE,col="blue")
```



`surface()`

- `curve` takes an expression and, as a side-effect, plots 1d curve by sweeping over `x`
- Suppose we want something like that but sweeping over two variables
- Let's build this as a good example of programming with functions and expressions
- Strategy: `surface()` should make `x` and `y` sequences, evaluate the expression at each combination to get `z`, and then call `persp()`—this is the function we used above, for the gradient descent 3d plot

First attempt at `surface()`

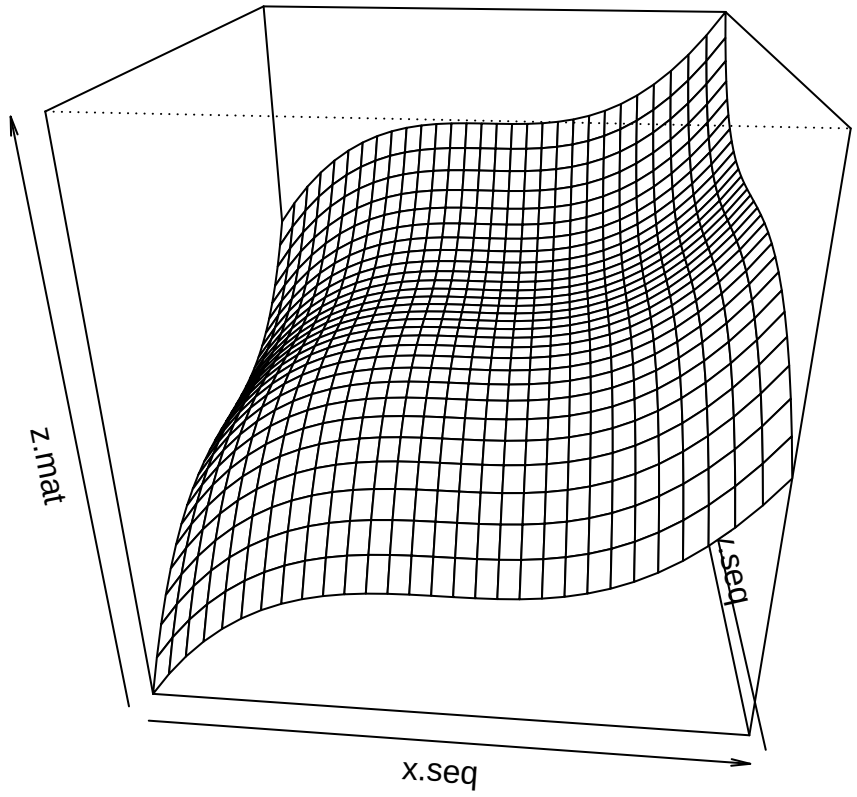
- Only works with vector-to-number functions:

```
surface.1 = function(f, from.x=0, to.x=1, from.y=0, to.y=1,
                     n.x=30, n.y=30, theta=5, phi=25, ...) {
  # Build the 2d grid
  x.seq = seq(from=from.x,to=to.x,length.out=n.x)
  y.seq = seq(from=from.y,to=to.y,length.out=n.y)
  plot.grid = expand.grid(x.seq,y.seq)
  z.vals = apply(plot.grid,1,f)
  z.mat = matrix(z.vals,nrow=n.x)

  # Plot with the persp function
  orig.mar = par()$mar # Save the original margins
  par(mar=c(1,1,1,1))  # Make the margins small
  r = persp(x.seq,y.seq,z.mat,theta=theta,phi=phi,...)
  par(mar=orig.mar) # Restore the original margins
```

```
  invisible(r)
}
```

```
surface.1(function(p){return(sum(p^3))},from.x=-1,from.y=-1)
```



Expressions and evaluation

- `curve()` doesn't require us to write a function every time; what's it's trick?
- **Expressions** are just another class of R object, so they can be created and manipulated
- One such manipulation is **evaluation**, as in

```
eval(expr,envir)
```

which evaluates the expression `expr` in the environment `envir`. The latter can be a data frame or a list

- If we were to type something like `x^2+y^2` as an argument to `surface.1`, then R tries to evaluate it prematurely
- `substitute()` returns the *unevaluted* expression
- `curve()` uses first `substitute(expr)` and then `eval(expr,envir)`,
  having made the right `envir`

Second attempt at `surface()`

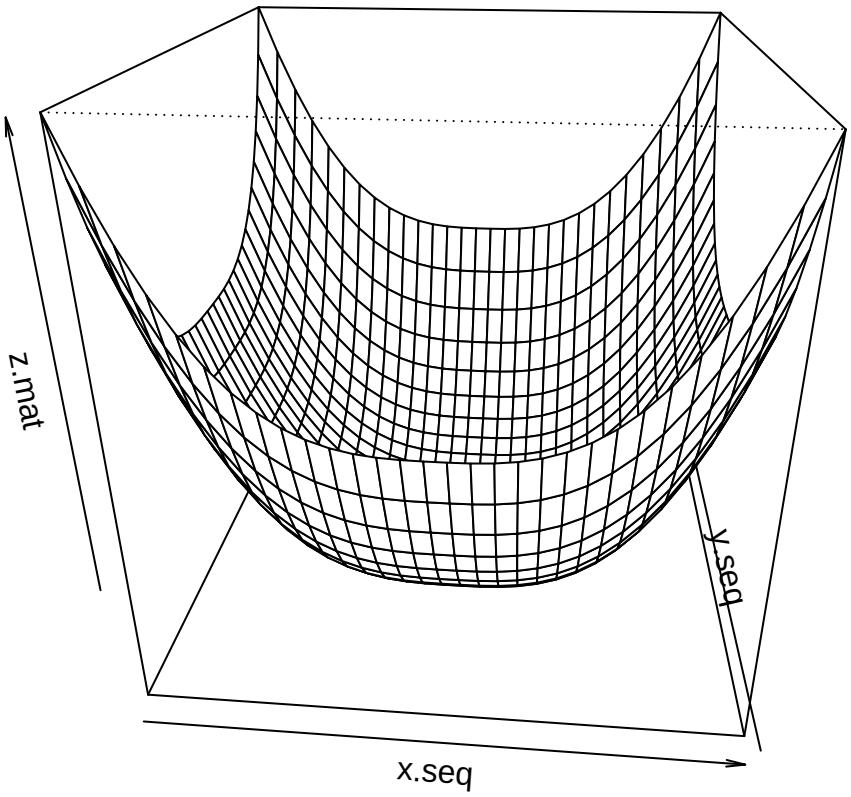
```
surface.2 = function(expr, from.x=0, to.x=1, from.y=0, to.y=1,
                     n.x=30, n.y=30, theta=5, phi=25, ...) {
   # Build the 2d grid
  x.seq = seq(from=from.x,to=to.x,length.out=n.x)
  y.seq = seq(from=from.y,to=to.y,length.out=n.y)
  plot.grid = expand.grid(x=x.seq,y=y.seq)

  # Evaluate the expression to get matrix of z values
  uneval.expr = substitute(expr)
  z.vals = eval(uneval.expr,envir=plot.grid)
  z.mat = matrix(z.vals,nrow=n.x)

  # Plot with the persp function
  orig.mar = par()$mar # Save the original margins
  par(mar=c(1,1,1,1))  # Make the margins small
  r = persp(x.seq,y.seq,z.mat,theta=theta,phi=phi,...)
  par(mar=orig.mar) # Restore the original margins
  invisible(r)
}
```

Now, easier to use!

```
surface.2(abs(x^3)+abs(y^3),from.x=-1,from.y=-1)
```



Summary

- In R, functions are objects, and can be arguments to other functions
  - Use this to do the same thing to many different functions
  - Separates writing the high-level operations and the first-order functions
  - Use `sapply` (etc.), wrappers, anonymous functions as adapters
- Functions can also be returned by other functions
  - Variables other than the arguments to the function are fixed by the environment of creation
  - Manipulating expressions lets us flexibly create functions