Lecture 16: Functions as Objects

36-350

20 October 2014

Previously. . .

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

Agenda

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

Reading: Sections 7.5, 7.11 and 7.13 of Matloff

Optional Recommended Reading: Chapter 3 of Chambers

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
- $\therefore$ 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

R Functions That Take Functions as Arguments

- `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*
- `ks.test()`: Compare *these* data to *this* cumulative distribution function
- `curve()`: Evaluate *this* function over *that* range, and plot the results

Some R Syntax Facts About Functions

- 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: 0x10698da98>
## <environment: namespace:base>
```

Some R Syntax Facts About Functions

- 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 R Syntax Facts About Functions

- Functions can be put into lists or even arrays
- A call to `function` returns a function object
  - body executed; access with `body(foo)`
  - arguments required: access with `formals(foo)`
    gives argument list of `foo`: names are argument names, values are expressions for defaults (if any)
  - parent environment: access with `environment(foo)`

Some R Syntax Facts About Functions

- 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`
- So far we've assigned that object to a name
- 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] 0.1192 0.2689 0.5000 0.7311 0.8808
```

Anonymous Functions

- Often handy when connecting other pieces of code
  - especially in things like `apply` and `sapply`
- Won't cluttering the workspace
- Can't be examined or re-used later

Example: grad()

- Problems in stats. come down to optimization
  So do lots of problems in econ., physics, CS, bio, ...
- Lots of optimization problems require the gradient of the **objective function**
- Gradient of f at x:

$$\nabla f(x) = \left[\left.\frac{\partial f}{\partial x_1}\right|_x \ldots \left.\frac{\partial f}{\partial x_p}\right|_x \right]$$

Example: grad()

- We do the same thing to get the gradient of f at x no matter what f is:

```
find the partial derivative of f with respect to each component of x
return the vector of partial derivatives
```

- It makes no sense to re-write this every time we change f!
- $\therefore$ 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

Example: grad()

From the package `numDeriv`

```r
grad(func, x, ...) # Plus other arguments
```

- Assumes `func` is a function which returns a single floating-point value
- Assumes `x` is a vector of arguments to `func`
  - If `x` is a vector and `func(x)` is also a vector, then it's assumed `func` is vectorized and we get a vector of derivatives
- Extra arguments in `...` get passed along to `func`
- Other functions in the package for the Jacobian of a vector-valued function, and the matrix of 2nd partials (Hessian)

Example: grad()

- Does it work as advertized?

```r
require("numDeriv")
```

```
## Loading required package: numDeriv
```

```r
just_a_phase <- runif(n=1,min=-pi,max=pi)
all.equal(grad(func=cos,x=just_a_phase),-sin(just_a_phase))
```

```
## [1] TRUE
```

```r
phases <- runif(n=10,min=-pi,max=pi)
all.equal(grad(func=cos,x=phases),-sin(phases))
```

```
## [1] TRUE
```

```r
grad(func=function(x){x[1]^2+x[2]^3}, x=c(1,-1))
```

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

Note: `grad` is perfectly happy with `func` being an anonymous function!

gradient.descent()

Now we can use this as a piece of a larger machine:

```r
gradient.descent <- function(f,x,max.iterations,step.scale,
  stopping.deriv,...) {
  for (iteration in 1:max.iterations) {
    gradient <- grad(f,x,...)
    if(all(abs(gradient) < stopping.deriv)) { break() }
    x <- x - step.scale*gradient
  }
  fit <- list(argmin=x,final.gradient=gradient,final.value=f(x,...),
    iterations=iteration)
  return(fit)
}
```

- Works equally well whether `f` is mean squared error of a regression, ψ error of a regression, (negative log) likelihood, cost of a production plan, . . .

Cautions

- *Scoping* `f` takes values for all names which aren't its arguments from the environment where it was defined, not the one where it is called (e.g., not from inside `grad` or `gradient.descent`)
- *Debugging* If `f` and `g` are both complicated, avoid debugging `g(f)` as a block; divide the work by writing *very simple* `f.dummy` to debug/test `g`, and debug/test the real `f` separately

Returning Functions: A trivial example

Functions can be return values like anything else

```r
make.noneuclidean <- function(ratio.to.diameter=pi) {
  circumference <- function(d) { return(ratio.to.diameter*d) }
  return(circumference)
}
```

Returning Functions: A trivial example (cont'd.)

```
try(circumference(10))
kings.i <- make.noneuclidean(3)
try(kings.i(10))
```

```
## [1] 30
```

```
formals(kings.i)
```

```
## $d
```

```
body(kings.i)
```

```
## {
##     return(ratio.to.diameter * d)
## }
```

```
environment(kings.i)
```

```
## <environment: 0x100fa9178>
```

```
try(circumference(10))
```

A Less Trivial Example

Create a linear predictor, based on sample values of two variables

```
make.linear.predictor <- function(x,y) {
  linear.fit <- lm(y~x)
  predictor <- function(x) {
   return(predict(object=linear.fit,newdata=data.frame(x=x)))
  }
  return(predictor)
}
```

The predictor function persists and works, even when the data we used to create it is gone

A Less Trivial Example

```
library(MASS); data(cats)
vet_predictor <- make.linear.predictor(x=cats$Bwt,y=cats$Hwt)
rm(cats)            # Data set goes away
vet_predictor(3.5)  # My cat's body mass in kilograms
```

```
##     1
## 13.76
```

A more mathematical example

- Instead of finding $\nabla f(x)$, find the function ∇f:

```
nabla <- function(f,...) {
  require("numDeriv")
  g <- function(x,...) { grad(func=f,x=x,...) }
  return(g)
}
```

Exercise: Write a test case!

Example: curve()

- You learned to use `curve` in the first week (because you did all of the assigned reading, including section 2.3.3 of the textbook)
- A call to `curve` looks like this:

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

`expr` is some expression involving a variable called `x`
which is swept `from` the value `a` `to` the value `b`
`...` are other plot-control arguments

- `curve` feeds the expression a vector `x` and expects a numeric vector back, e.g.

  ```
  curve(x^2 * sin(x))
  ```

  is fine

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)
```

Try this! Also try

```
curve(psi(x=10,c=x),from=-20,to=20)
```

and explain it to yourself

Using curve() with our own functions

- If our function doesn't take vectors to vectors, `curve` becomes unhappy

```r
mse <- function(y0,a,Y=gmp$pcgmp,N=gmp$pop) {
   mean((Y - y0*(N^a))^2)
}
```

```
> curve(mse(a=x,y0=6611),from=0.10,to=0.15)
Error in curve(mse(a = x, y0 = 6611), from = 0.1, to = 0.15) :
  'expr' did not evaluate to an object of length 'n'
In addition: Warning message:
In N^a : longer object length is not a multiple of shorter object length
```

How do we solve this?

Using curve() with our own functions

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

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

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

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

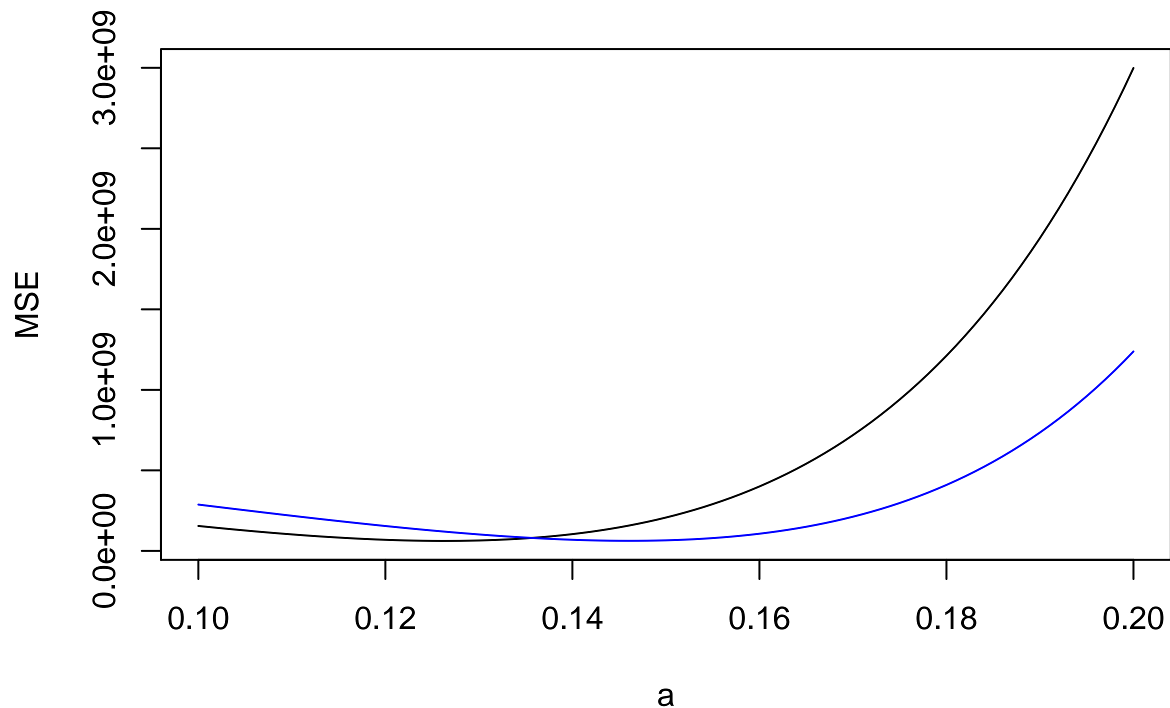
```
## [1] 154701953
```

```r
mse.plottable <- function(a,...){ return(sapply(a,mse,...)) }
mse.plottable(seq(from=0.10,to=0.15,by=0.01),y0=6611)
```

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

Using curve() with our own functions

```r
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")
```



Using curve() with our own functions

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

```
mse.vec <- Vectorize(mse, vectorize.args=c("y0","a"))
mse.vec(a=seq(from=0.10,to=0.15,by=0.01),y0=6611)
```

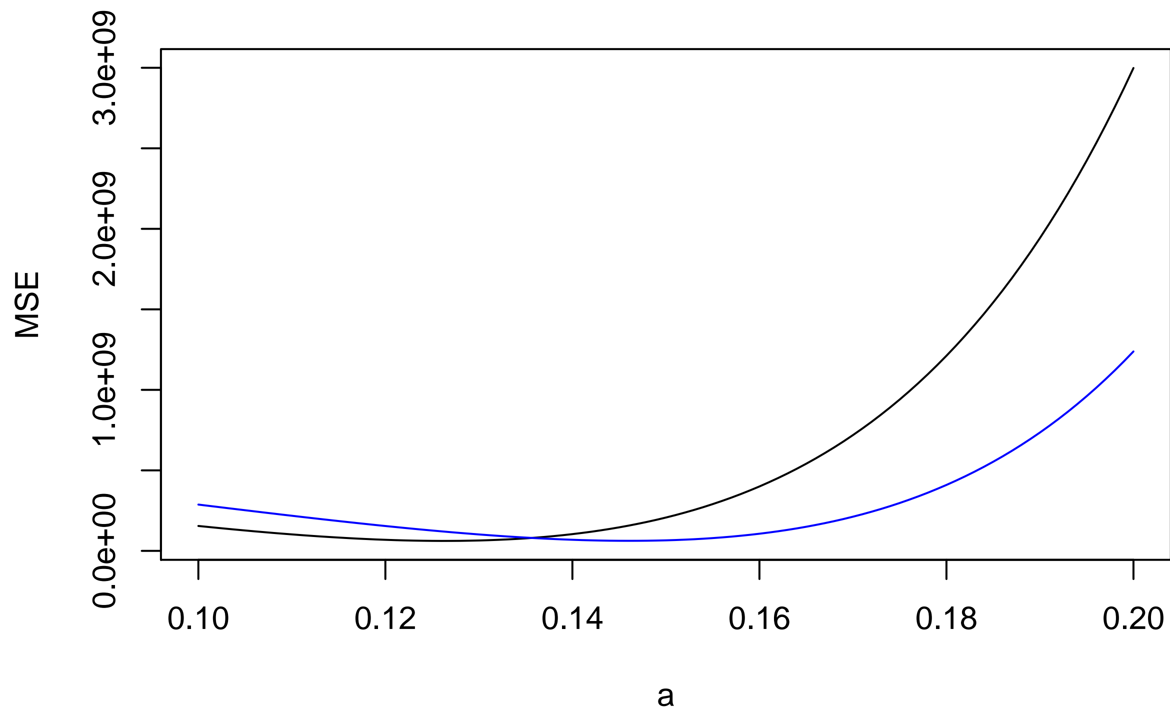
```
## [1] 154701953 102322974  68755654  64529166 104079527 207057513
```

```
mse.vec(a=1/8,y0=c(5000,6000,7000))
```

```
## [1] 134617132  74693733  63732256
```

Using curve() with our own functions

```
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")
```



Example: surface()

- `curve` takes an expression and, as a side-effect, plots a 1-D curve by sweeping over `x`
- Suppose we want something like that but sweeping over two variables
- Built-in plotting function `contour`:

  ```
  contour(x,y,z, [[other stuff]])
  ```

  `x` and `y` are vectors of coordinates, `z` is a matrix of the corresponding shape
  (see `help(contour)` for graphical options)
- Strategy: `surface` should make `x` and `y` sequences, evaluate the expression at each combination to get `z`, and then call `contour`

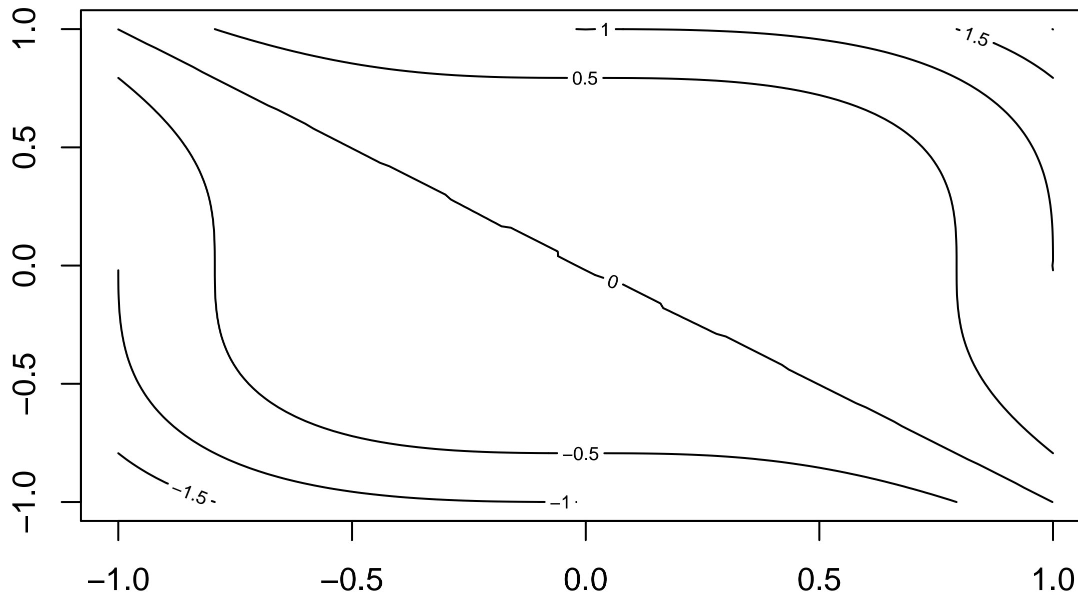
First attempt at surface()

- 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=101,
  n.y=101,...) {
  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)
  z.values <- apply(plot.grid,1,f)
  z.matrix <- matrix(z.values,nrow=n.x)
  contour(x=x.seq,y=y.seq,z=z.matrix,...)
  invisible(list(x=x.seq,y=y.seq,z=z.matrix))
}
```

First attempt at surface()

```
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 manipulation is **evaluation**

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

evaluates the expression `expr` in the environment `envir`, which can be a data frame or even just a list

- When we type something like `x^2+y^2` as an argument to `surface.1`, 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`

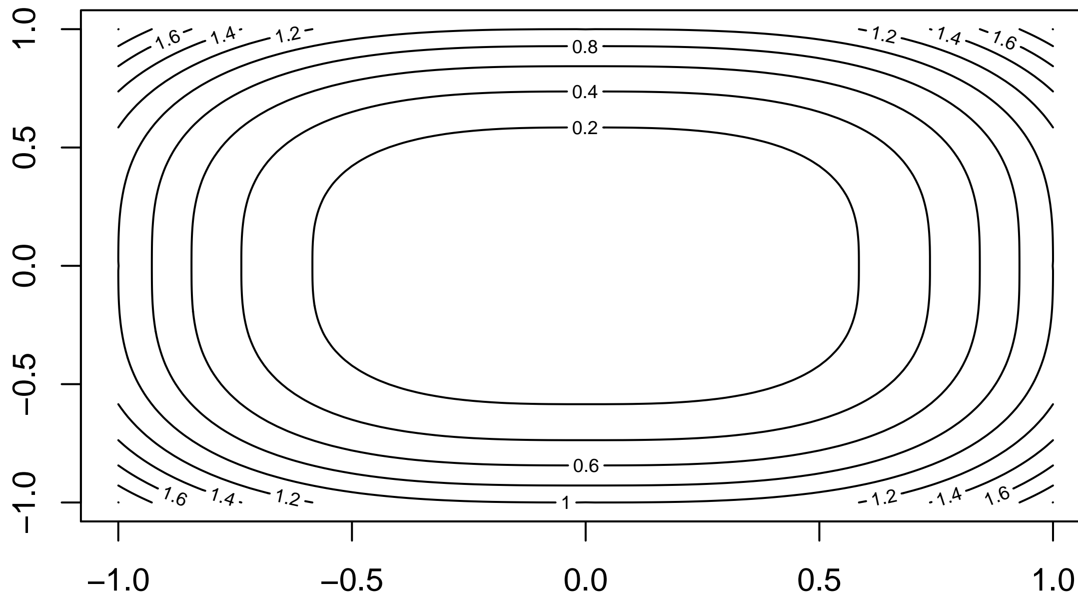
Second attempt at surface()

```
surface.2 <- function(expr,from.x=0,to.x=1,from.y=0,to.y=1,n.x=101,
  n.y=101,...) {
  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)
```

```
  unevaluated.expression <- substitute(expr)
  z.values <- eval(unevaluated.expression,envir=plot.grid)
  z.matrix <- matrix(z.values,nrow=n.x)
  contour(x=x.seq,y=y.seq,z=z.matrix,...)
  invisible(list(x=x.seq,y=y.seq,z=z.matrix))
}
```

Second attempt at surface()

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



Evaluating at Combinations

- Evaluating a function at every combination of two arguments is a really common task
- There is a function to do it for us: `outer`

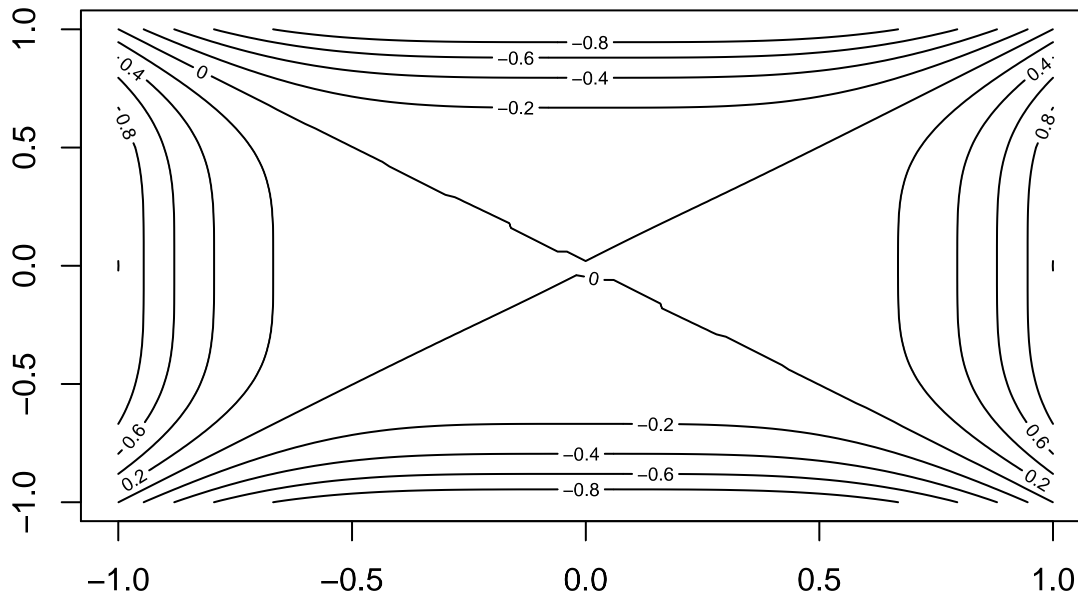
Third attempt at surface()

```
surface.3 <- function(expr,from.x=0,to.x=1,from.y=0,to.y=1,n.x=101,
  n.y=101,...) {
  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)
  unevaluated.expression <- substitute(expr)
  z <- function(x,y) {
    return(eval(unevaluated.expression,envir=list(x=x,y=y)))
  }
```

```
  z.values <- outer(X=x.seq,Y=y.seq,FUN=z)
  z.matrix <- matrix(z.values,nrow=n.x)
  contour(x=x.seq,y=y.seq,z=z.matrix,...)
  invisible(list(x=x.seq,y=y.seq,z=z.matrix, func=z))
}
```

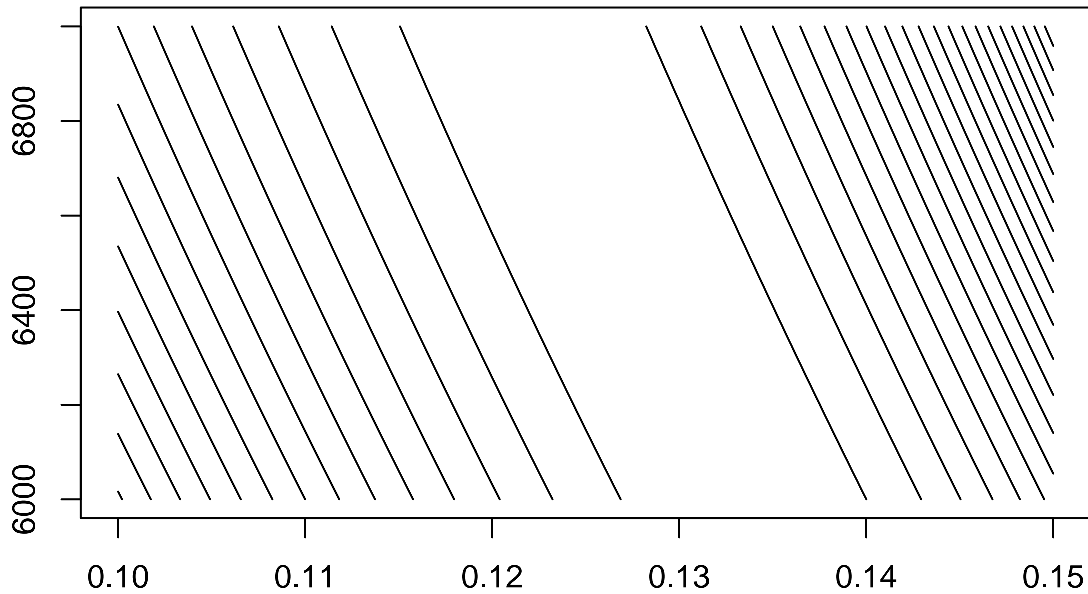
Third attempt at surface()

```
surface.3(x^4-y^4,from.x=-1,from.y=-1)
```



surface()

```
surface.3(mse.vec(a=x,y0=y),from.x=0.10,to.x=0.15,
          from.y=6e3,to.y=7e3,nlevels=20)
```



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

Functions of Functions: Mathematically

- Maximum, and location of the maximum: takes f, gives number

$$\max_{x} f(x) \ , \ \mathrm{argmax}_x f(x)$$

- Derivative of f at x_0: takes a function and a point, gives a number

$$\frac{df}{dx}(x_0) \equiv \lim_{h \rightarrow 0} \frac{f(x_0 + h) - f(x_0)}{h}$$

- Definite integral of f over $[a, b]$: takes a function and two points, gives a number

$$\int_{a}^{b} f(x) dx \equiv \lim_{n \rightarrow \infty} \sum_{i=0}^{n-1} \left(\frac{b-a}{n}\right) f\left(a + i\frac{b-a}{n}\right)$$

Mathematical view cont'd.

- Functions of functions which return numbers sometimes are sometimes called **functionals**, e.g., expectation values:

$$\mathbb{E}[f(X)] \equiv \int_{\text{all } x} f(x)p(x)dx$$

- $\nabla f(x_0)$ takes f and x_0, gives vector: not strictly a functional
- ∇f is another, vector-valued function
  ∇ takes a function and returns a function
  ∇ is an **operator**, not a functional

Mathematically

- Something which takes a function in and gives a function back is an **operator**
- Differentiation: the operator d/dx takes f and gives a new function
- Gradient: the operator ∇ takes f and gives a new function
  similarly $\nabla\cdot$, $\nabla\times$, ...
- Indefinite integration: $\int_{-\infty}^{x} f(u)du$ takes f and gives a new function
- Fourier transform: takes f and gives a new function

$$\tilde{f}(\omega) = \int_{x=-\infty}^{x=\infty} f(x)e^{2i\pi\omega x}dx$$

Bonus: Writing Our Own gradient()

- Suppose we didn't know about the `numDeriv` package..

-Use the simplest possible method: change `x` by some amount, find the difference in `f`, take the slope `method="simple"` option in `numDeriv::grad`

- Start with pseudo-code

```
gradient <- function(f,x,deriv.steps) {
  # not real code
  evaluate the function at x and at x+deriv.steps
  take slopes to get partial derivatives
  return the vector of partial derivatives
}
```

Bonus Example: gradient()

A naive implementation would use a `for` loop

```
gradient <- function(f,x,deriv.steps,...) {
  p <- length(x)
  stopifnot(length(deriv.steps)==p)
  f.old <- f(x,...)
  gradient <- vector(length=p)
  for (coordinate in 1:p) {
   x.new <- x
   x.new[coordinate] <- x.new[coordinate]+deriv.steps[coordinate]
   f.new <- f(x.new,...)
   gradient[coordinate] <- (f.new - f.old)/deriv.steps[coordinate]
  }
  return(gradient)
}
```

Works, but it's so repetitive!

Bonus Example: gradient()

Better: use matrix manipulation and `apply`

```
gradient <- function(f,x,deriv.steps,...) {
  p <- length(x)
  stopifnot(length(deriv.steps)==p)
  x.new <- matrix(rep(x,times=p),nrow=p) + diag(deriv.steps,nrow=p)
  f.new <- apply(x.new,2,f,...)
  gradient <- (f.new - f(x,...))/deriv.steps
  return(gradient)
}
```

(clearer, and half as long)

- Presumes that `f` takes a vector and returns a single number
- Any extra arguments to `gradient` will get passed to `f`
- Check: Does this work when `f` is a function of a single number?

Bonus Example: gradient()

- Acts badly if `f` is only defined on a limited domain and we ask for the gradient somewhere near a boundary
- Forces the user to choose `deriv.steps`
- Uses the same `deriv.steps` everywhere, imagine $f(x) = x^2 \sin x$

... and so on through much of a first course in numerical analysis (or at least sec. 5.7 of *Numerical Recipes*)