Functions as Objects

James M. Flegal

Agenda

- Functions are objects: can be arguments for or returned by other functions
- `curve()`
- `tidyverse`

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
- 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
- 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

Functions as Objects

- We have already seen examples
- `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

Syntax Facts About Functions

- Typing a function's name, without parentheses, in the terminal gives you its source code
- Functions are their own **class** in R

```
class(sin)
```

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

```
class(sample)
```

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

```
resample <- function(x) { sample(x, size=length(x), replace
class(resample)
```

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

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)`
  - parent environment: access with `environment(foo)`

Syntax Facts About Functions

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

```r
typeof(resample)
```

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

```r
typeof(sample)
```

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

```r
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] 0.1192029 0.2689414 0.5000000 0.7310586 0.8807971
```

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()

- Many problems in statistics come down to optimization
- So do lots of problems in economics, physics, CS, biology, ...
- 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 and return the vector of partial derivatives
- It makes no sense to re-write this every time we change f!
- 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`

```
grad(func, x, ...)
```

- 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()

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

```
## Loading required package: numDeriv

## Warning: package 'numDeriv' was built under R version 4.0.2
```

```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
```

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

gradient.descent()

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

```
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

- Functions can be return values like anything else
- 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

Returning Functions

```r
library(MASS)
```

```
## Warning: package 'MASS' was built under R version 4.0.2
```

```r
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
## 13.76256
```

Example: curve()

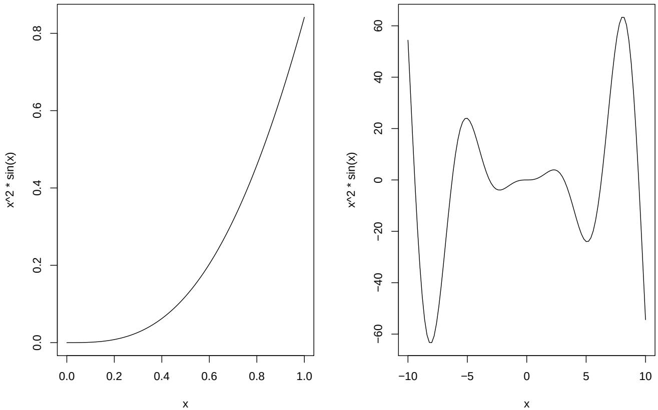
- A call to `curve` looks like this:

```
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 (so this is fine)

Example: curve()

```
par(mfrow = c(1, 2))
curve(x^2 * sin(x))
curve(x^2 * sin(x), from=-10, to=10)
```

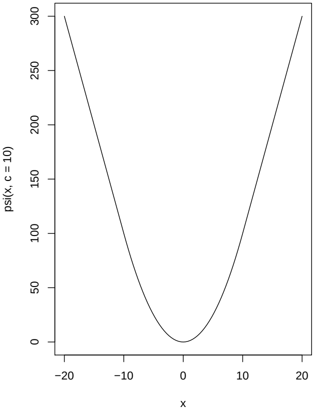


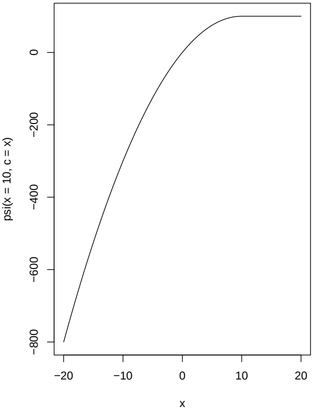

```
par(mfrow = c(1, 1))
```

Example: curve()

- Can use our own function in `curve`

```
par(mfrow = c(1, 2))
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)
curve(psi(x=10,c=x),from=-20,to=20)
```






```
par(mfrow = c(1, 1))
```

Example: curve()

- Unhappy if our function doesn't take vectors to vectors

```
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
```

- Note the code chunk above contains `eval=FALSE`
- How do we solve this?

Example: curve()

- 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)
```

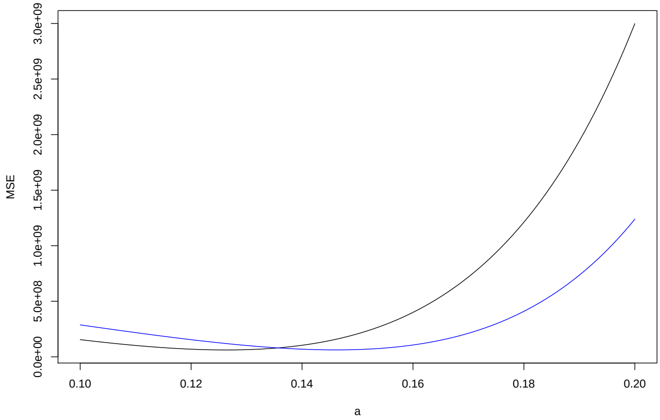
```
## [1] 154701953
```

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

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

Example: curve()

```
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")
```



Example: curve()

- 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)
```

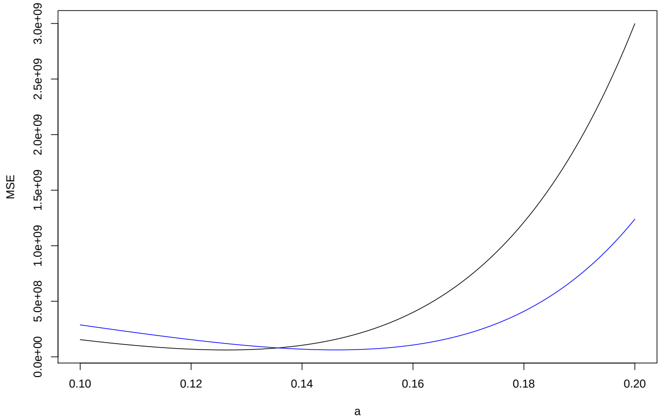
```
## [1] 154701953 102322974  68755654  64529166 104079527 207057513
```

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

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

Example: curve()

```
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")
```



`tidyverse`

- The `tidyverse` is a collection of open source R packages that "share an underlying design philosophy, grammar, and data structures" of tidy data
- Includes `ggplot2`, `dplyr`, `tidyr`, `readr`, `purrr`, `tibble`, `stringr`, and `forcats`, which provide functionality to model, transform, and visualize data.
- Written by Hadley Wickham and others and promoted by RStudio
- Detractors argue it is more complicated to learn for beginners or non-programmers

tidyverse

- https://www.tidyverse.org
- Advanced R
- Could spend the rest of the quarter exploring the `tidyverse` ...
- But our focus is **Computational Statistics**

Computational Statistics

- Optimization
- Simulations
- Monte Carlo Methods
- Bootstrap
- Cross-Validation
- Density Estimation
- Bayesian Statistics
- Markov Chain Monte Carlo
- Permutation Tests