Writing Functions (Advanced)

Canhong Wen

Agenda

- Example: Parameter estimation code
- Multiple functions
- Recursion: Making hard problems simpler

A Complex Example: Fitting a Model

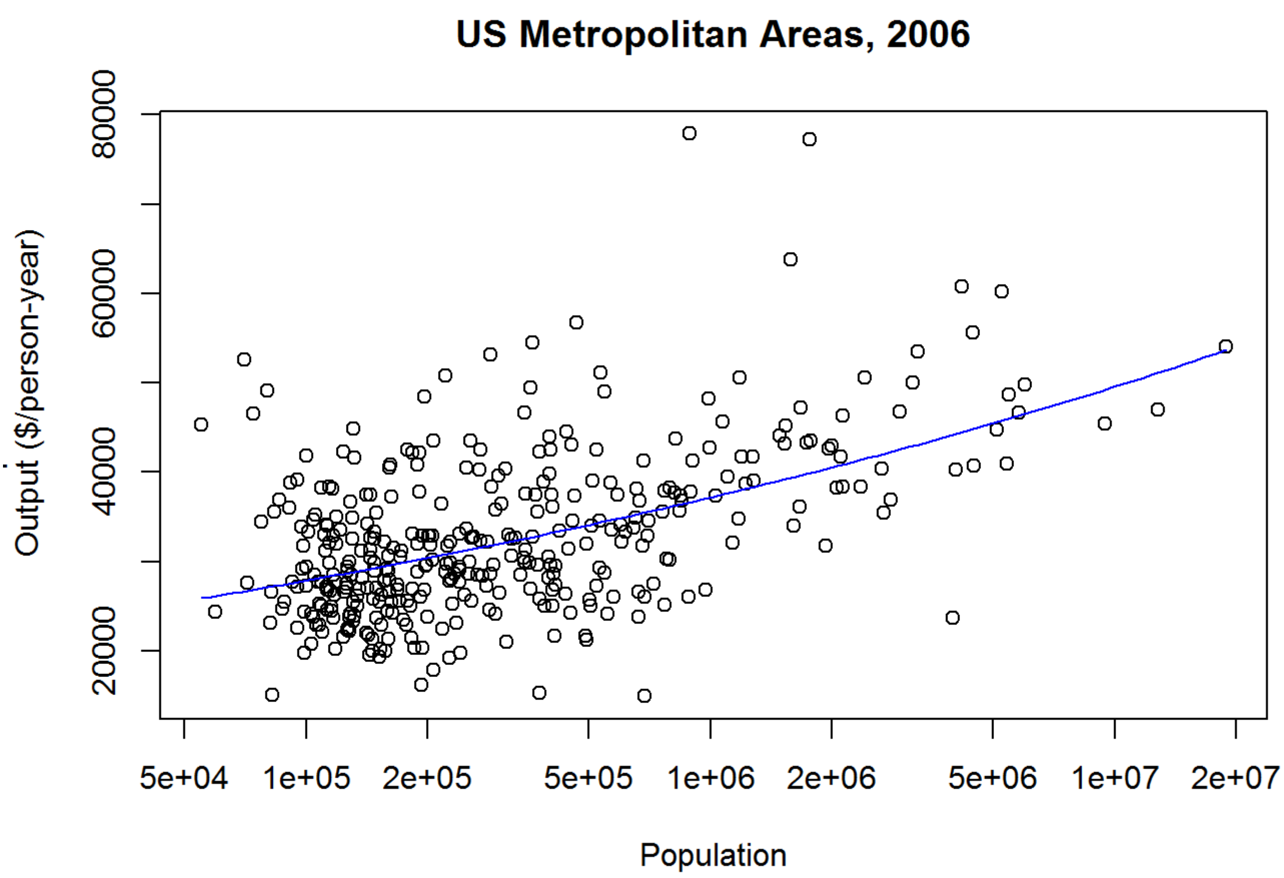
Fact: bigger cities tend to produce more economically per capita

A proposed statistical model (Geoffrey West et al.):

$$Y = y_0 N^{a} + \mathrm{noise}$$

where Y is the per-capita "gross metropolitan product" of a city, N is its population, and y_0 and a are parameters

```
gmp <- read.table("data/gmp.dat")
gmp$pop <- gmp$gmp/gmp$pcgmp
plot(pcgmp~pop, data=gmp, log="x", xlab="Population", ylab="Per-Capita Economic
     Output ($/person-year)", main="US Metropolitan Areas, 2006")
curve(6611*x^(1/8),add=TRUE,col="blue")
```



Fitting a Model

$$Y = y_0 N^{a} + \mathrm{noise}$$

Take $y_0=6611$ for now and estimate a by minimizing the mean squared error

Approximate the derivative of error w.r.t a and move against it
$$\begin{eqnarray*} MSE(a) & \equiv & \frac{1}{n}\sum_{i=1}^{n}{(Y_i - y_0 N_i^a)^2}\\ MSE^{\prime}(a) & \approx & \frac{MSE(a+h) - MSE(a)}{h}\\ a_{t+1} - a_{t} & \propto & -MSE^{\prime}(a) \end{eqnarray*}$$

An actual first attempt at code:

```
maximum.iterations <- 100
deriv.step <- 1/1000
step.scale <- 1e-12
stopping.deriv <- 1/100
iteration <- 0
deriv <- Inf
a <- 0.15
while ((iteration < maximum.iterations) && (deriv > stopping.deriv)) {
  iteration <- iteration + 1
  mse.1 <- mean((gmp$pcgmp - 6611*gmp$pop^a)^2)
  mse.2 <- mean((gmp$pcgmp - 6611*gmp$pop^(a+deriv.step))^2)
  deriv <- (mse.2 - mse.1)/deriv.step
  a <- a - step.scale*deriv
}
list(a=a,iterations=iteration,converged=(iteration < maximum.iterations))
```

```
## $a
## [1] 0.1258166
##
## $iterations
## [1] 58
##
## $converged
## [1] TRUE
```

What’s wrong with this?

- Not *encapsulated*: Re-run by cutting and pasting code — but how much of it? Also, hard to make part of something larger
- *Inflexible*: To change initial guess at a, have to edit, cut, paste, and re-run
- *Error-prone*: To change the data set, have to edit, cut, paste, re-run, and hope that all the edits are consistent
- *Hard to fix*: should stop when *absolute value* of derivative is small, but this stops when large and negative. Imagine having five copies of this and needing to fix same bug on each.

Will turn this into a function and then improve it

First attempt, with logic fix:

```
estimate.scaling.exponent.1 <- function(a) {
  maximum.iterations <- 100
  deriv.step <- 1/1000
  step.scale <- 1e-12
  stopping.deriv <- 1/100
  iteration <- 0
  deriv <- Inf
  while ((iteration < maximum.iterations) && (abs(deriv) > stopping.deriv)) {
    iteration <- iteration + 1
    mse.1 <- mean((gmp$pcgmp - 6611*gmp$pop^a)^2)
    mse.2 <- mean((gmp$pcgmp - 6611*gmp$pop^(a+deriv.step))^2)
    deriv <- (mse.2 - mse.1)/deriv.step
    a <- a - step.scale*deriv
  }
  fit <- list(a=a,iterations=iteration,
    converged=(iteration < maximum.iterations))
  return(fit)
}
```

Problem: All those magic numbers!

Solution: Make them defaults

```
estimate.scaling.exponent.2 <- function(a, y0=6611,
  maximum.iterations=100, deriv.step = .001,
  step.scale = 1e-12, stopping.deriv = .01) {
  iteration <- 0
  deriv <- Inf
  while ((iteration < maximum.iterations) && (abs(deriv) > stopping.deriv)) {
    iteration <- iteration + 1
    mse.1 <- mean((gmp$pcgmp - y0*gmp$pop^a)^2)
    mse.2 <- mean((gmp$pcgmp - y0*gmp$pop^(a+deriv.step))^2)
    deriv <- (mse.2 - mse.1)/deriv.step
    a <- a - step.scale*deriv
  }
  fit <- list(a=a,iterations=iteration,
    converged=(iteration < maximum.iterations))
  return(fit)
}
```

Problem: Why type out the same calculation of the MSE twice?
Solution: Declare a function

```
estimate.scaling.exponent.3 <- function(a, y0=6611,
  maximum.iterations=100, deriv.step = .001,
  step.scale = 1e-12, stopping.deriv = .01) {
  iteration <- 0
  deriv <- Inf
  mse <- function(a) { mean((gmp$pcgmp - y0*gmp$pop^a)^2) }
  while ((iteration < maximum.iterations) && (abs(deriv) > stopping.deriv)) {
    iteration <- iteration + 1
    deriv <- (mse(a+deriv.step) - mse(a))/deriv.step
    a <- a - step.scale*deriv
  }
  fit <- list(a=a,iterations=iteration,
    converged=(iteration < maximum.iterations))
  return(fit)
}
```

`mse()` declared inside the function, so it can see `y0`, but it's not added to the global environment

Problem: Locked in to using specific columns of `gmp`; shouldn't have to re-write just to compare two data sets
Solution: More arguments, with defaults

```
estimate.scaling.exponent.4 <- function(a, y0=6611,
  response=gmp$pcgmp, predictor = gmp$pop,
  maximum.iterations=100, deriv.step = .001,
  step.scale = 1e-12, stopping.deriv = .01) {
  iteration <- 0
  deriv <- Inf
  mse <- function(a) { mean((response - y0*predictor^a)^2) }
  while ((iteration < maximum.iterations) && (abs(deriv) > stopping.deriv)) {
    iteration <- iteration + 1
    deriv <- (mse(a+deriv.step) - mse(a))/deriv.step
    a <- a - step.scale*deriv
  }
  fit <- list(a=a,iterations=iteration,
    converged=(iteration < maximum.iterations))
  return(fit)
}
```

Respecting the interfaces: We could turn the `while()` loop into a `for()` loop, and nothing outside the function would care

```
estimate.scaling.exponent.5 <- function(a, y0=6611,
  response=gmp$pcgmp, predictor = gmp$pop,
  maximum.iterations=100, deriv.step = .001,
  step.scale = 1e-12, stopping.deriv = .01) {
  mse <- function(a) { mean((response - y0*predictor^a)^2) }
  for (iteration in 1:maximum.iterations) {
    deriv <- (mse(a+deriv.step) - mse(a))/deriv.step
    a <- a - step.scale*deriv
    if (abs(deriv) <= stopping.deriv) { break() }
  }
  fit <- list(a=a,iterations=iteration,
    converged=(iteration < maximum.iterations))
  return(fit)
}
```

What have we done?

The final code is shorter, clearer, more flexible, and more re-usable

Exercise: Run the code with the default values to get an estimate of \(a\); plot the curve along with the data points

Exercise: Randomly remove one data point — how much does the estimate change?

Exercise: Run the code from multiple starting points — how different are the estimates of \(a\)?

Exercise: Run the code with the default values to get an estimate of \(a\); plot the curve along with the data points

```
a <- 0.15
estimate.scaling.exponent.5(a)
```

```
## $a
## [1] 0.1258166
##
## $iterations
## [1] 58
##
## $converged
## [1] TRUE
```

Exercise: Randomly remove one data point — how much does the estimate change?

```
set.seed(1)
idx <- sample(nrow(gmp),1)
gmp1 <- gmp[-idx,]
estimate.scaling.exponent.5(a, response=gmp1$pcgmp, predictor = gmp1$pop)
```

```
## $a
## [1] 0.125855
##
## $iterations
## [1] 58
##
## $converged
## [1] TRUE
```

Exercise: Run the code from multiple starting points — how different are the estimates of \(a\)?

```
estimate.scaling.exponent.5(a = 0)
```

```
## $a
## [1] 0.1258166
##
## $iterations
## [1] 79
##
## $converged
## [1] TRUE
```

```
estimate.scaling.exponent.5(a = 0.1)
```

```
## $a
## [1] 0.1258166
##
## $iterations
## [1] 62
##
## $converged
## [1] TRUE
```

```
# estimate.scaling.exponent.5(a = 0.2, deriv.step = 5e-3)
```

How We Extend Functions

- Multiple functions: Doing different things to the same object

- Sub-functions: Breaking up big jobs into small ones

Why Multiple Functions?

Meta-problems:

- You've got more than one problem
- Your problem is too hard to solve in one step
- You keep solving the same problems

Meta-solutions:

- Write multiple functions, which rely on each other
- Split your problem, and write functions for the pieces
- Solve the recurring problems once, and re-use the solutions

Writing Multiple Related Functions

Statisticians want to do lots of things with their models: estimate, predict, visualize, test, compare, simulate, uncertainty, …

Write multiple functions to do these things

Make the model one object; assume it has certain components. See `lm`, `coef.lm`, `predict.lm`, etc.

```
x <- rnorm(15)
y <- x + rnorm(15)
fit <- lm(y ~ x)
class(fit)
```

```
## [1] "lm"
```

```
all(predict(lm(y ~ x))==predict.lm(lm(y ~ x)))
```

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

Consistent Interfaces

- Functions for the same kind of object should use the same arguments, and presume the same structure
- Functions for the same kind of task should use the same arguments, and return the same sort of value (to the extent possible)

Keep related things together

- Put all the related functions in a single file or in a single filefolder (by building a project).
- Source them together
- Use comments to note ***dependencies***

Power-Law Scaling

Remember the model:

$$Y = y_0 N^{a} + \mathrm{noise}$$ $$(\mathrm{output}\ \mathrm{per}\ \mathrm{person}) = $$ $$ (\mathrm{baseline}) (\mathrm{population})^{\mathrm{scaling}\ \mathrm{exponent}} + \mathrm{noise}$$

Estimated parameters a, y_0 by minimizing the mean squared error

Exercise: Modify the estimation code from last time so it returns a list, with components `a` and `y0`

Predicting from a Fitted Model

Predict values from the power-law model:

```
# Predict response values from a power-law scaling model
# Inputs: fitted power-law model (object), vector of values at which to make
  # predictions at (newdata)
# Outputs: vector of predicted response values
predict.plm <- function(object, newdata) {
  # Check that object has the right components
  stopifnot("a" %in% names(object), "y0" %in% names(object))
  a <- object$a
  y0 <- object$y0
  # Sanity check the inputs
  stopifnot(is.numeric(a),length(a)==1)
  stopifnot(is.numeric(y0),length(y0)==1)
  stopifnot(is.numeric(newdata))
  return(y0*newdata^a)  # Actual calculation and return
}
```

Plotting from a Fitted Model

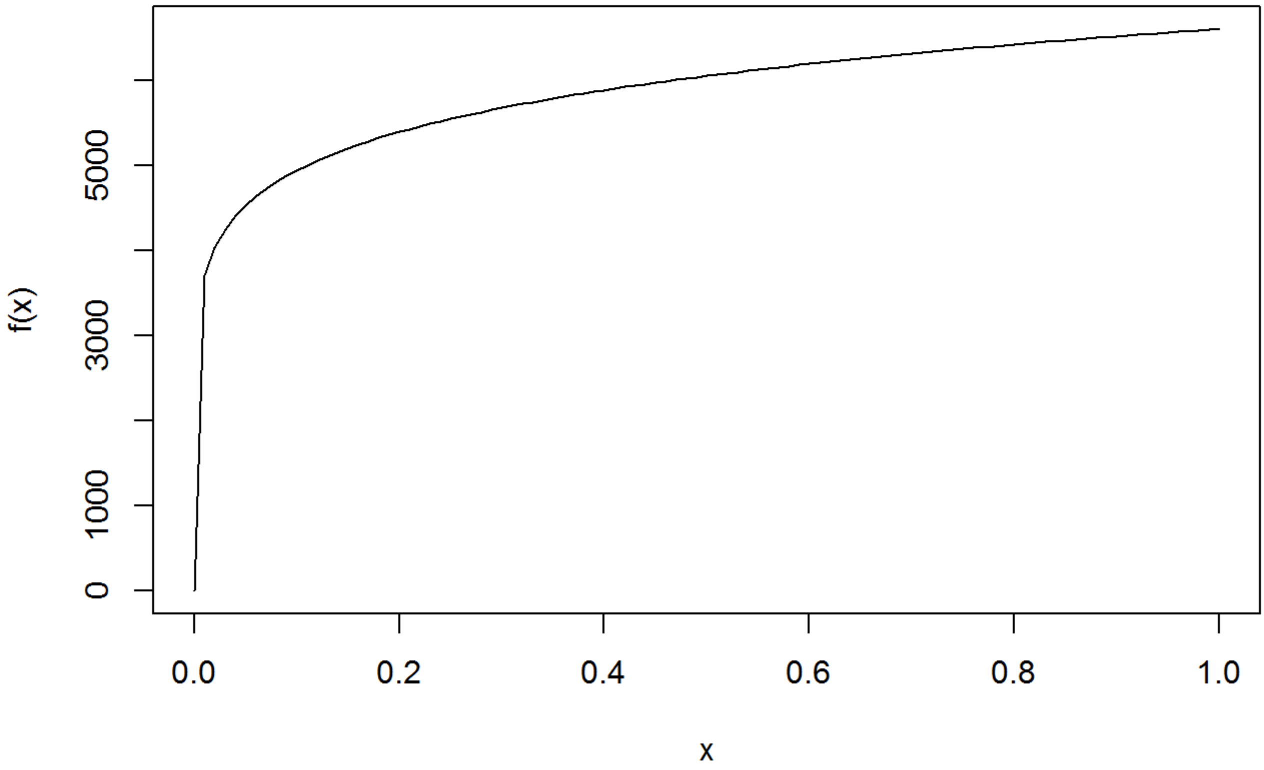
```
# Plot fitted curve from power law model over specified range
# Inputs: list containing parameters (plm), start and end of range (from, to)
# Outputs: TRUE, silently, if successful
# Side-effect: Makes the plot
plot.plm.1 <- function(plm,from,to) {
  # Take sanity-checking of parameters as read
  y0 <- plm$y0 # Extract parameters
  a <- plm$a
  f <- function(x) { return(y0*x^a) }
  curve(f(x),from=from,to=to)
  # Return with no visible value on the terminal
  invisible(TRUE)
}
```

Plotting from a Fitted Model

When one function calls another, use as a meta-argument, to pass along unspecified inputs to the called function:

```
plot.plm.2 <- function(plm,...) {
  y0 <- plm$y0
  a <- plm$a
  f <- function(x) { return(y0*x^a) }
  # from and to are possible arguments to curve()
  curve(f(x), ...)
  invisible(TRUE)
}
```

```
fit <- estimate.scaling.exponent.5(0.15)
fit$y0 <- 6611
plot.plm.2(fit)
```



Sub-Functions

Solve big problems by dividing them into a few sub-problems

- Easier to understand, get the big picture at a glance
- Easier to fix, improve and modify
- Easier to design
- Easier to re-use solutions to recurring sub-problems

Rule of thumb: A function longer than a page is probably too long

Sub-Functions or Separate Functions?

Defining a function inside another function

- Pros: Simpler code, access to local variables, doesn't clutter workspace
- Cons: Gets re-declared each time, can't access in global environment (or in other functions)
- Alternative: Declare the function in the same file, source them together

Rule of thumb: If you find yourself writing the same code in multiple places, make it a separate function

Plotting a Power-Law Model

Our old plotting function calculated the fitted values

But so does our prediction function

```
plot.plm.3 <- function(plm,from,to,n=101,...) {
  x <- seq(from=from,to=to,length.out=n)
  y <- predict.plm(object=plm,newdata=x)
  plot(x,y,...)
  invisible(TRUE)
}
```

Recursion

Reduce the problem to an easier one of the same form:

```
my.factorial <- function(n) {
  if (n == 1) {
    return(1)
  } else {
    return(n*my.factorial(n-1))
  }
}
```

Recursion

Or multiple calls (Fibonacci numbers (https://en.wikipedia.org/wiki/Fibonacci_number)):

```
fib <- function(n) {
  if ( (n==1) || (n==0) ) {
   return(1)
  } else {
   return (fib(n-1) + fib(n-2))
  }
}
```

Exercise: Convince yourself that any loop can be replaced by recursion; can you always replace recursion with a loop?

Summary

- **Functions** bundle related commands together into objects: easier to re-run, easier to re-use, easier to combine, easier to modify, less risk of error, easier to think about
- **Interfaces** control what the function can see (arguments, environment) and change (its internals, its return value)
- **Calling** functions we define works just like calling built-in functions: named arguments, defaults
- **Multiple functions** let us do multiple related jobs, either on the same object or on similar ones
- **Sub-functions** let us break big problems into smaller ones, and re-use the solutions to the smaller ones
- Recursion (https://en.wikipedia.org/wiki/Recursion_(computer_science)) is a powerful way of making hard problems simpler